DISS. ETH NO. 18622

Reasoning About Data Abstraction in Contract Languages

A dissertation submitted to the
Swiss Federal Institute of Technology Zurich
(ETH Zurich)

for the degree of
Doctor of Sciences

presented by

Ádám Péter Darvas
Dipl. Technical Informatics
Budapest University of Technology and Economics

born February 18, 1979
citizen of Hungary

accepted on the recommendation of

Prof. Peter Müller, examiner
Prof. Reiner Hähnle, co-examiner
Prof. Gary T. Leavens, co-examiner

2009

Abstract

Due to the large and ever increasing complexity of software systems, abstraction plays an important role in the process of software development. Abstraction is also essential in the formal specification of programs because it allows one to write specifications in an implementation-independent way, which is indispensable for information hiding and facilitates readability and maintainability of specifications.

State-of-the-art specification languages provide powerful means of abstraction that are natural to use for programmers. However, previous work provided only partial solutions for reasoning about specifications that make use of these means. This thesis presents techniques that allow one to reason about two means of abstraction that are commonly used in object-oriented specification languages: pure methods and model classes.

Pure methods are side-effect free methods of a program. As such, specification languages allow them to be called in specification expressions. In order to reason about calls to pure methods, the methods have to be encoded and axiomatized in the underlying logic of the verification environment. The encoding is non-trivial if pure methods are considered to be weakly-pure, that is, if they are not completely side-effect free and are allowed to allocate, initialize, and return new objects. Such state changes are observable in specification expressions, thus an encoding has to take them into account. The axiomatization of pure methods has to be done with care because unsatisfiable, contradicting, or ill-founded specifications can lead to an inconsistent axiom system if specifications are blindly turned into axioms.

This thesis proposes a practical encoding of weakly-pure methods as well as an axiomatization technique that poses proof obligations on specifications that guarantee the consistency of the axiom system that is extracted from the specifications.

Model classes are classes that are used only for specification purposes and provide object-oriented interfaces for essential mathematical concepts, such as sets or relations. Specifications can be written in an abstract way by expressing properties in terms of model classes and their operations. A promising approach to reason about specifications that make use of model classes is to map the classes and their operations to the built-in structures and functions of the underlying theorem prover of the verification environ-

ment. However, this can lead to unsound reasoning if there is a mismatch between the properties of a model class and the properties of the structure to which the class is mapped.

This thesis presents a technique that formally checks if there is semantic correspondence between a model class and the structure it is mapped to. The approach not only enables reasoning about programs specified in terms of model classes but also helps in writing better specifications for model classes and in identifying and checking redundant specifications.

The contributions of this thesis are independent of any particular specification language, verification technique, and theorem prover. Thus, the results are applicable for a wide range of program-verification tools.

Zusammenfassung

Aufgrund der erheblichen und stetig anwachsenden Komplexität von Software-Systemen spielt Abstraktion im Prozess der Software-Entwicklung eine bedeutende Rolle. Abstraktion ist auch für die formale Spezifikation von Programmen zentral, weil dadurch Spezifikationen auf eine implementierungs-unabhängige Art und Weise geschrieben werden können. Dies ist für das Geheimnisprinzip unabdingbar und erhöht die Lesbarkeit sowie die Wartbarkeit der Spezifikationen.

Moderne Spezifikationssprachen bieten ausdruckstarke Mittel für die Abstraktion und erlauben dem Programmierer einen unkomplizierten Gebrauch dieser Mittel. Bis anhin vorliegende Arbeiten brachten nur Teillösungen für die Beweisführung über Spezifikationen, die von diesen Mitteln Gebrauch machen, hervor. Die vorliegende Arbeit zeigt Techniken auf, welche die Beweisführung über zwei Mittel der Abstraktion erlauben, welche in objektorientierten Spezifikationssprachen häufig gebraucht werden: pure Methoden und Modell-Klassen.

Pure Methoden sind nebeneffektfreie Methoden eines Programmes. Aufgrund der Nebeneffektfreiheit der Methoden erlauben Spezifikationssprachen Aufrufe solcher Methoden in Spezifikationen. Damit ein Beweis über solche Methoden geführt werden kann, müssen die Methoden in der zugrundeliegenden Logik der Beweisführung kodiert sowie axiomatisiert werden. Die Kodierung ist jedoch nicht trivial, wenn pure Methoden schwach pur sind. Das heisst, diese sind nicht vollständig nebeneffektfrei sondern erlauben das Erzeugen, Initialisieren sowie Zurückgeben von neuen Objekten. Weil Zustandsänderungen in der Spezifikation beobachtbar sind, müssen diese bei der Kodierung beachtet werden. Die Axiomatisierung von puren Methoden muss vorsichtig vollzogen werden, weil Spezifikationen, die widersprüchlich, nicht erfüllbar oder nicht wohldefiniert sind, zu inkonsistenten Axiomationssystemen führen können.

Die vorliegende Doktorarbeit schlägt sowohl eine praktisch zu handhabende Kodierung von schwach puren Methoden als auch eine Technik der Axiomatisierung vor. Diese Technik formuliert Beweisobligationen für Spezifikationen, welche die Konsistenz des Axiomsystems garantiert, das aus den Spezifikation extrahiert wird.

Modell-Klassen sind Klassen, welche ausschliesslich für Spezifikationen

gebraucht werden und objektorientierte Schnittstellen für grundlegende mathemathische Konzepte, wie beispielsweise Mengen und Relationen, darstellen. Spezifikationen können auf eine abstrakte Art und Weise geschrieben werden, indem Eigenschaften in Form von Modell-Klassen und deren Operationen formuliert werden. Ein vielversprechender Lösungsansatz für die Beweisführung über Spezifikationen, welche Modell-Klassen gebrauchen, ist die Zuordnung von Klassen und Operationen zu bereits bestehende Strukturen und Funktionen. Dies kann jedoch zu falschen Beweisen führen, wenn eine Diskrepanz zwischen den Eigenschaften einer Modell-Klasse und der Struktur, der die Klasse zugeordnet ist, vorliegt.

Die vorliegende Doktorarbeit zeigt eine Technik auf, welche überprüft, ob eine semantische Übereinstimmung zwischen einer Modell-Klasse und einer Struktur, der die Klasse zugeordnet ist, vorliegt. Dieser Ansatz erlaubt nicht nur die Beweisführung über Programme, die in Form von Modell-Klassen spezifiziert sind, sondern hilft auch bei der Verbesserung der Spezifikation von Modell-Klassen sowie bei der Identifizierung und Prüfung von redundanten Spezifikationen.

Der Beitrag dieser Doktorarbeit ist unabhängig von einer spezifischen Spezifikationssprache, der Überprüfungstechnik sowie dem Theorem-Beweiser und die Ergebnisse sind für eine Vielzahl an Programm-Beweisern anwendbar.

Acknowledgments

Getting to the phase of writing acknowledgments marks the very end of a long professional and personal journey, which was accompanied by many colleagues and friends to whom I would like to express my sincere gratitude.

First and foremost, I thank my supervisor, Prof. Peter Müller, for his constant guidance, support, and encouragement, which were all essential for the completion of my thesis. I am positive that his supervision did not only lead to my professional but also to my personal growth. His dedication to research and his way of interacting with the members of our group created a very impulsive, open, and enjoyable working environment and atmosphere.

I would like to thank my colleague, Arsenii Rudich, for his invaluable contribution to the material presented in Part I of my thesis. Our co-operation was always extremely instructive for me.

I am grateful to my co-examiners, Prof. Reiner Hähnle and Prof. Gary T. Leavens, for their careful reading of and detailed comments on my thesis. Gary's enduring commitment to JML always affected me in a stimulating way during my PhD work. I am indebted to Reiner for kindly offering me to join the KeY group at the Chalmers University of Technology before my studies at ETH. It was under his supervision when I really understood what program verification was all about, thereby pushing me even more towards this exciting topic. My stay at Chalmers would not have been possible without the financial support of Prof. David Sands.

It was an honor to me to work with K. Rustan M. Leino during my internship at Microsoft Research, Redmond. His supervision and extraordinary personality made those three months an unforgettable experience. I also thank Mike Barnett for his cordiality and for helping me out whenever I was snagged with the Spec# tool.

The countless technical discussions with Farhad Mehta, Vijay D'Silva, Prof. Burkhart Wolff, and Stefan Berghofer considerably inspired and fostered my work. I am grateful to Peter H. Schmitt for referring me to the concept of theory interpretation as the proper "tool" for developing the technique presented in Part II of my thesis.

I thank my students Ghislain Fourny, Erich Laube, Marcello Miragliotta, Adrian Moos, Yoshimi Takano, and Geraldine von Roten for their committed work, which was of great help in the completion of my thesis.

I am glad that I had the opportunity to work at the Chair of Programming Methodology; a research group consisting of such diverse and unique characters. For their friendship and for the countless fun hours we spent together over the past years, I would like to thank Werner M. Dietl, Hermann Lehner, Martin Nordio, Arsenii Rudich, and Joseph N. Ruskiewicz. For all the memorable times spent together, I also thank the members of the Chair of Software Engineering: Till Bay, Ilinca Ciupa, Piotr Nienaltowski, Michela Pedroni, and Bernd Schoeller.

Special thank goes to the friends who spent so many amusing hours with me outside of ETH. The friendship of Vijay D'Silva, Yvonne Gahler, Ilaria Galli, Monika Kast, Andreas Leitner, Julian Lorenz, Farhad Mehta, Yvonne Moh, Francis Moss, and Almut Scherer turned out to be at least as important to me as my PhD. I am immensely grateful to Moni for being such a caring neighbor and for her heroic work on translating my abstract to German.

Finally, I want to express my sincere gratitude to my "hinterland" in Hungary, without which it would have been impossible for me to carry on for so many years abroad. I thank my family, Péter, Ágnes, and Enikő, for their continuous support and encouragement over the last three decades; Zsófia Virág and Zsóka Vásárhelyi for their patience and fondness; and Fanni, who makes me the happiest uncle by always recognizing me and by always being willing to play hide-and-seek with me despite my infrequent visits during my PhD studies.

Contents

List of Figures

Chapter 1

Introduction

Software rules our world and, thus, our daily life. No matter whether buying a cup of coffee, listening to music, looking at a monthly bank-account statement, or driving a car; in some way, some kind of software is most likely involved. As a consequence, the quality of software that surrounds us has a great impact on our lives.

The impact may be different though. Getting a green tee instead of the desired espresso might make you loose 2 Franks and say a few informal words, but after all, your day is not ruined (unless you are allergic to green tee and did not notice the mistake in time). On the other hand, a faulty braking system may cause severe damage to the car, and injury to the passengers and others.

Thus, the software of systems that may cause significant economic loss or may endanger human lives, have to be of outstanding quality. Since the size and complexity of such systems can be enormous, it is typically impossible to assure their high quality by means of traditional software engineering, such as testing or code reviewing. Therefore, in areas where the importance of software quality is beyond the importance of all other objectives, the use of formal program verification is gaining more and more attention.

Formal program verification shows by mathematical means that the program at hand does what it is supposed to do. To achieve this goal, the program first needs to be specified in a formal way to give a precise description of its expected behavior. Once equipped with specifications, the program needs to be formally verified to show that the implementation corresponds to the specification.

In order to make formal program verification successful on programs developed in the industry, many different techniques and tools need to be developed and combined. For instance, techniques to express alias control, methodologies to handle concurrent programs, and automated as well as interactive theorem provers that work well in different domains, such as first-order logic or Presburger arithmetic.

This thesis focuses on one particular aspect of formal program verification, namely, on the development of techniques that support the verification of common means of abstraction used in state-of-the-art object-oriented specification languages.

1.1 Abstraction in Program Specifications

The word *abstraction* comes from the Latin *ab* (away from) and *trahere* (to draw). Abstraction is the process of hiding certain characteristics of some concept or object in order to keep only its essential characteristics. What characteristics get hidden and what kept, depends on the particular purpose of the abstraction. The goal is to reduce complexity of the domain at hand in order to enable focusing on the relevant details.

Abstraction proved to be an important notion in Computer Science and is present in several different areas and levels. As stated by Guttag [51], abstraction plays an important role in the building of complex and large software systems, too:

> *[...] the amount of complexity that the human mind can cope with at any instant in time is considerably less than that embodied in much of the software that one might wish to build. Thus the key problem in the design and implementation of large software systems is reducing the amount of complexity or detail that must be considered at any one time. One way to do this is via the process of abstraction.*

Note that the quote is from 1977. Today, the size and complexity of software systems is typically several magnitudes larger than back in the seventies. And this growing tendency is expected to continue in the years to come. Thus, abstraction plays an ever increasing role in the process of software development.

Abstraction also plays an important role in the formal specification of programs. Abstraction allows one to write specifications in an implementation independent way, which is indispensable for information hiding, and promotes readability and maintainability of specifications.

1.1.1 Abstraction in Design Specifications

Several specification languages and approaches have been developed for the design and modeling of computing systems. Some of the most well-known are VDM-SL [68], Z [129], RAISE/RSL [113], the B-Method [1], Event-B [2, 3], and UML/OCL [131, 109]. Specifications written in such languages are typically high-level abstractions of the actual realizations of the systems.

Still, specifications written on this level can already reveal, for instance, misunderstandings, contradicting assumptions, or uncovered cases. To reduce the gap between the abstract design and the actual realization, approaches such as the B-Method [1] have been developed to support refinement of specifications.

Common to these design specification languages is that they come with a *mathematical vocabulary* that allows one to specify properties of the main functionalities of programs. For instance, VDM-SL provides a number of built-in types such as integers and booleans, and simple data types such as sets, lists, and mappings, together with operations on these data types. Z, the B-Method, and Event-B are based on set theory from which a rich mathematical vocabulary is derived, containing structures such as trees and sequences. OCL provides built-in primitive types as well as types that model mathematical structures such as sets, bags, and sequences.

Equipped with such mathematical means, one can specify properties of computer programs in a programming language and implementation independent way. For example, a specification of a banking software could maintain a set of accounts and a mapping from accounts to the set of holders of a given account. Operations could then, for instance, extend the set of accounts by new ones or update the mapping to include a new relation between an account and a holder.

In the course of verification, the relation between the abstract description and the concrete implementation is given by *abstraction functions*, as proposed by Hoare [62]. For instance, an abstraction function defined over an array used in the implementation may yield a mathematical set containing the elements of the array. Alternatively, if the order of the elements matter for the purpose of the abstraction, the function could yield a sequence.

1.1.2 Abstraction in BISLs

Behavioral interface specification languages (BISLs) specify both the interfaces and the behavior of programs. Interface specifications provide information on the interfaces between program components [135, 52]. For instance, in an object-oriented language this may be information on the name of methods available in a given type, the arity of those methods, the types of the arguments and the return value, and the types of exceptions the method may throw. Behavior specifications provide information on the expected behavior of types and their methods, expressed by invariants and method specifications. Method specifications typically consist of pre- and postconditions and frame properties.

BISLs can be divided into two groups: *two-tiered* BISLs use two different languages for the specification of programs. One language is specifically tailored to a programming language and the other language is independent of any programming language [52]. On the other hand, *one-tiered* BISLs

use only one language for the specification of programs, and that language is designed for a particular target programming language.

1.1.2.1 Abstraction in Two-tiered Specification Languages

The term "two-tiered specification language" first appeared in Wing's thesis in 1983 [134], and the approach was pioneered by Larch [52]. *Larch interface languages* are specifically designed for the underlying programming language. Several Larch interface languages have been developed, such as LCL for Standard C, LM3 for Modula-3, and Larch/Smalltalk for Smalltalk. The *Larch Shared Language* (LSL) is independent of the underlying programming language and defines the vocabulary that is to be used for the specification of behavior.

The basic building blocks of the LSL are *traits*. Traits introduce new sorts and operators, and define them by axioms. Even elementary symbols like `/\` and `+` for logical conjunction and mathematical addition are introduced by traits. Traits may also introduce abstract data types such as lists, maps, and relations.

In the specification of behavior, abstraction is achieved by the use of operators and data types introduced by the traits. Users may extend the set of traits provided by the LSL, which facilitates the building of domain-specific abstractions. Similar to design languages, the connection between abstract specifications and actual implementations is established by (implicit) abstraction functions.

Using the elements of LSL for abstraction has great advantages. (1) Since the semantics of LSL is typically simpler than that of the underlying programming languages, the chance of making mistakes in specifications is smaller. (2) Since the semantic gap between LSL and logic is smaller than the semantic gap between programming languages and logic, formal reasoning becomes simpler.

On the other hand, the disadvantage of using Larch, and two-tiered specification languages in general, is that users need to get acquainted with the mathematical vocabulary of the specification language. This may be problematic for programmers who have little or no experience with mathematical formalisms. In fact, one of the common reasons why programmers reject the use of formal methods is that they are not willing to learn a separate language just for specifying their programs, in particular, if that language is fundamentally different from that of the programming language.

1.1.2.2 Abstraction in One-tiered Specification Languages

A one-tiered specification language merges the two tiers by specifying both the interface and the behavior of programs using one language, which is based on the target programming language. Thereby, the syntax of spec-

ification expressions becomes nearly identical to that of the programming language, allowing programmers to learn the specification language within a few hours or days. This possibly makes programmers more willing to write specifications and accept formal methods.

The first one-tiered specification language was the Java Modeling Language (JML) [73, 74, 77], which was designed for the specification of Java programs. In JML, the programming-language independent component (*i.e.*, LSL in Larch) is substituted with a collection of Java types that model mathematical structures. Thereby, this collection gives the mathematical vocabulary of JML.

Remark. Languages that follow the idea of Design by Contract (DbC) proposed by Meyer [94, 95] also use one single language for the specification of programs. Still, we only consider such languages to be one-tiered if they provide a mathematical vocabulary to facilitate abstraction. For instance, in our view, Eiffel [96, 97], the first language that implemented DbC, became a one-tiered language when a mathematical vocabulary was developed for it in 2006 [123].

While one-tiered specification languages are simpler to learn and use for programmers than two-tiered languages, they pose challenges for program specification and verification. The specification of programs becomes more difficult because the semantics of the underlying programming language has to be taken into account, too. Verification becomes harder because the semantic gap between specification expressions and the logic used for formal reasoning gets larger, and this gap has to be bridged in a way that preserves both the original semantics of specification expressions and the soundness of reasoning.

In one-tiered specification languages, even the means of abstraction has to be expressed on the level of the underlying programming language. Therefore, the approach of using mathematical vocabularies, such as the LSL, is not directly applicable. This lead to the introduction of other means of abstraction. Three of the most important means are pure methods, model classes, and model fields [31].

Pure Methods. The concept of *procedures* is one of the crucial elements of Structured Programming [40]. David Gries defines its main use as follows [49]:

> *The main use of the procedure is in abstraction. [...] The main property that we single out, once a procedure is written, is what it does; the main property that we omit from consideration is how it does it.*

That is, the caller of a procedure need not have any knowledge about the implementation details of the procedure. The only thing that matters for the caller is the functionality of the procedure. Furthermore, procedures are a unit of reuse: using procedure calls avoids the duplication of code.

One-tiered languages carry this idea and advantage over to method calls that appear in specification expressions [31, 32]. The meaning of a method call is given by the method's specification and thus the specification does not have to be repeated. This avoids duplication of the specification and provides means to specify behavior without mentioning implementation details.

The use of method calls in specifications can be thought of as a replacement for the application of functions and predicates that mathematical vocabularies of two-tiered specification languages provide. Accordingly, methods that are called in specifications are required to behave like functions and predicates do: they have to be deterministic and side-effect free [77].

Model Classes. Model classes provide the interfaces, specification, and often the implementation of data types such as sets and sequences. Model classes can be thought of as direct replacements for the mathematical structures that two-tiered specification languages provide. Accordingly, model classes have to behave like mathematical structures do: they have to be immutable and contain only side-effect free operations [77].

Model Fields. Model fields are similar to ordinary fields except that they are to be used only for specification purposes. Thus, they may not occur in implementations. Model fields provide a means of abstraction: their values are determined by the values of non-model fields as specified by user-defined relations. In particular, these relations may involve model classes, which allows one to express program states in terms of mathematical structures.

Remark. In the sequel, we do not discuss other means of abstraction (for instance, data groups [80] for the modular specification of frame conditions) as the main focus of the thesis is on pure methods and model classes.

Example 1.1. Figure 1.1 demonstrates the use of the three means of abstraction through a class that represents a bank account. The example is written in JML, thus specifications are embedded in Java comments that start with an at-sign (`@`). Implementations are omitted.

Class `Account` declares two private fields, `holders` and `balance`, in order to keep track of the holders of an account and the current balance, respectively. Class `Person` represents the clients of the bank, and thus the holders of accounts. `Person`'s interface and implementation are omitted.

```
import java.util.*;
//@ import org.jmlspecs.models.JMLObjectSet;

public class Account {
  private ArrayList holders;
  private long balance;

  //@ public model JMLObjectSet _set;
  //@ private represents _set <-
  //@   new JMLObjectSet { Person h | holders.contains(h) && true };
  //@ public invariant !_set.isEmpty();
  //@ public invariant getBalance() >= 0;

  /*@ private normal_behavior
    @   ensures \result == balance;
    @*/
  public /*@ pure @*/ long getBalance() { ... }

  /*@ public normal_behavior
    @   ensures \result == _set.int_size();
    @*/
  public /*@ pure @*/ int numOfHolders() { ... }

  /*@ public normal_behavior
    @   ensures \result <==>
    @           (getBalance() >= 100000 && _set.int_size() <= 2);
    @*/
  public /*@ pure @*/ boolean isPremium() { ... }

  /*@ public normal_behavior
    @   requires !_set.has(h);
    @   ensures (\forall Person o;
    @             _set.has(o) <==>
    @               (\old(_set.has(o)) || o.equals(h)));
    @*/
  public void addHolder(Person h) { ... }

  /*@ public normal_behavior
    @   requires isPremium();
    @   ensures getBalance() == \old(getBalance()) * 1.02;
    @*/
  public void addBonus() { ... }

  // other fields, invariants, constructors, and methods are omitted
}
```

Figure 1.1: An abstract specification of class `Account`

Class `Account` declares three query methods: `getBalance` returns the balance of the account, `numOfHolders` yields the number of holders of an account, and `isPremium` queries whether the account is a premium account or not. These methods are side-effect free, which is specified by the keyword **`pure`** in their signatures. Specification expressions may contain calls (only) to pure methods.

The class declares two mutating methods: `addHolder` adds a new holder to an account and `addBonus` increases the balance of an account by some amount.

The specification of the class declares a public model field `_set`, which represents the holders in an abstract way, in the form of a mathematical set. This is done by using a pre-defined model class of JML, `JMLObjectSet`. The model class represents a set of objects. The abstraction function that determines the value of the model field is given by the **`represents`** clause. It is private, as it mentions the private field `holders`. The abstraction function uses a set comprehension to collect all `Person`-instances that the array list of the concrete representation contains.

The class contains two invariants. The first invariant expresses that an account must have at least one holder. This is expressed through the public model field, and thus the invariant can be declared as public, too. Note that pure method `isEmpty` is called on the model field, that is, the method is called on an instance of model class `JMLObjectSet`. The second invariant expresses that the balance of an account may not be negative. The invariant is expressed by the use of the public pure method `getBalance`, thus the invariant can also be declared as public.

Method `getBalance` has a private specification because it refers to the private `balance` field. All other methods have implementation independent specifications as they make use of the public pure methods and the public model field. Therefore, these specifications can be declared as public. For instance, the precondition of method `addHolder` expresses that the `Person`-instance passed as parameter is not in the set of account holders yet, and the postcondition expresses that the set of account holders has been extended by the instance, leaving all other holders intact.

As the example demonstrates, the use of pure methods (both that of the model class and that of `Account`) in specifications is just as natural and powerful as in program text. The use of pure methods provides a means of information hiding, and makes specifications more concise and comprehensible. Furthermore, the use of pure methods increases maintainability of specifications because even if the meaning (that is, specification and implementation) of a pure method changes over time, the specification of methods that use such pure-method calls need not be modified. For instance, the criteria of an account being premium or not might change over time. Still, the specification of method `addBonus` can remain unchanged. □

1.2 Problems and Contributions

As mentioned above, the main advantage of a one-tiered specification language is that programmers can easily learn its use since the syntax and semantics of the specification language is close to that of the programming language. However, this causes difficulties for formal verification because the semantic gap increases between the specification language and the logic used for reasoning.

The main contribution of this thesis is the development of techniques that fill this gap. In this section, we highlight the issues that such techniques have to cope with and summarize the main contributions of the thesis. The highlighted issues are of concern for every approach that supports a one-tiered specification language, such as Eiffel, JML, and Spec# [9, 10]. The developed techniques can be adopted by verification techniques or tools that use first-order logic in their prover back-end, such as ESC/Java2 [70], Jack [26], Jive [98], Krakatoa [91], the Spark Toolset [7], and Spec#.

Encoding of Specification Expressions

In order to reason about programs, user-specified specification expressions have to be encoded in the language of the underlying logic. As expressed by the misgivings of Leavens in [72] while arguing against C++ expressions to appear in Larch/C++ specifications, such an encoding is non-trivial:

> *To ensure that the assertions are well-defined, they can't call C++ code. (If an assertion used C++ code, one couldn't easily interpret expressions used in assertions as abstract values, because C++ code doesn't deal with mathematical entities.)*

Below we refine these issues into three categories and point out some of the fundamental problems that the design of an encoding has to face.

Well-definedness of Expressions. Programs may contain expressions whose execution possibly results in a program crash or in throwing an exception. Typical examples include the dereferencing of `null` and division by zero. Clearly, such expressions indicate programming mistakes.

In one-tiered specification languages such expressions may occur in specifications, too. However, if we think of formal specifications as mathematical text, then such expressions are mathematically unreasonable.

One way to overcome this problem is to filter out such specifications before further analyzing the program at hand. However, it is not straightforward how this can be done. In particular, it is not obvious what has to be shown and what information may be used during the filtering process.

Handling Object Allocation. Specifications may only contain side-effect free expressions. Therefore, method calls are only allowed to occur in specification expressions if the corresponding callees are declared to be pure. The literature proposes to encode pure methods by uninterpreted function symbols, and pure-method calls by applications of the corresponding functions [32, 35, 65, 33]. The approach seems to be justified since pure methods have to be side-effect free (and deterministic, as discussed later), just like mathematical functions.

However, pure methods are not completely free of side effects. It proved to be useful to allow pure methods to allocate, initialize, and return new objects [103, 118]. Therefore, either an encoding of pure methods has to take into account state changes that are potentially observable or restrictions must be made on specification expressions such that the state changes become non-observable. No matter which option is taken, the solution has to be carefully worked out in order to prevent semantical mismatches and unsoundness.

Preserving Semantics. Modern programming languages come with well-defined semantics. A natural requirement for an encoding is to preserve the semantics prescribed by the language at hand. Otherwise, the verification process would attempt to prove properties that are different from the specified properties [27]. This would undermine the whole purpose of verification by possibly leading to unexpected or unsound results. Results that would also differ from the outcome of runtime assertion checking.

As a simple example, consider the commonly used "short-circuit" operator `||`. Although it expresses logical disjunction, its semantics is not identical to logical disjunction that we know from classical logic.

Contributions. We propose an encoding of specification expressions in classical two-valued first-order logic.

We address the issue of ill-defined expressions by formally proving that each assertion can be evaluated to either **true** or **false**. The technique filters out bogus specification expressions, which will be rejected. Thereby, the technique is capable of catching errors in specifications. Furthermore, in contrast to other techniques (*e.g.*, underspecification), it leads to a semantics that is close to that of runtime assertion checking. The technique is well-studied for first-order logic. We adapt it to one-tiered specification languages and prove that the adapted variant is an instance of the one developed for first-order logic.

We demonstrate through examples that state changes made by pure methods are observable. We present an encoding that takes state changes into account by explicitly modeling them. We illustrate the drawbacks of such an explicit modeling and propose an encoding that yields significantly

simpler formulas. The simplified encoding considers pure methods to be completely side-effect free. We precisely define the conditions under which the simplified encoding preserves the original semantics, and we define means that ensure that the conditions hold.

The proposed encoding together with the elimination of ill-defined expressions guarantees that the semantics of specification expressions is preserved by the encoding.

Axiomatization of Pure Methods

As mentioned above, pure-method calls are encoded by applications of uninterpreted functions. In order to reason about formulas that contain such function applications, meaning has to be assigned to these function symbols. This is typically done by stating axioms over the function symbols based on the specifications of the corresponding pure methods.

However, it would be dangerous to blindly turn user-defined specifications into axioms. Postconditions may be infeasible, the specification of multiple pure methods may contradict each other, or (mutually) recursive specifications may be ill-founded. An axiom system that is extracted from such specifications may be inconsistent, thereby potentially leading to unsound reasoning.

Therefore, proper checks must be performed on specifications before axioms are emitted. However, it is not trivial what these checks should be and what information may be assumed when performing these checks, in particular, when specifications contain (possibly mutually recursive) calls to pure methods.

Contributions. We propose a technique that guarantees the consistency of the axiom system that is extracted from user-defined specifications. The technique poses proof obligations to ensure the feasibility of pure-method specifications as well as well-foundedness for recursive specifications.

The technique deals with dependencies between method calls. That is, if the specification of some pure method n contains a call to another pure method m, then the consistency of m is attempted to be proven first, and, in case of success, the properties of m can be used when attempting to prove the consistency of n. This way the tracking of dependencies gives a means to precisely define which parts of the specifications may be used at what points of the axiomatization process. Furthermore, the development of the technique reveals that certain dependencies should be forbidden, that is, calls to pure methods should not be allowed in certain specification constructs.

We demonstrate that well-definedness checking and consistency checking are closely related and should not be divided into two independent tasks. The proposed technique combines the two in a natural way.

Handling of Model Classes

Model classes only contain pure methods, thus the encoding and axiomatization techniques described above apply to the methods of model classes. However, applying the same techniques for model classes may not be desirable. Several theorem provers come with a set of built-in theories containing essential mathematical structures. The tactics of theorem provers are optimized for such built-in theories rather than encodings of model classes via newly introduced function symbols.

Therefore, the literature proposes to map model classes and their methods directly to the built-in theories and symbols of the theorem prover at hand [29, 75, 76]. This way, methods of model classes need not be axiomatized. Instead, calls to such methods are encoded as applications of the corresponding predicate and function symbols.

Obviously, crucial to the soundness of this technique is to ensure that the mapping is correct. That is, the semantics of related model classes and mathematical structures, and related methods and symbols have to match. Otherwise, static program verifiers might produce results that come unexpected for programmers who rely on the model-class specification. Furthermore, the results may also vary between different theorem provers, which define certain operations slightly differently. Moreover, if model classes are implemented and the implementation is based on the model-class specification, then the outcome of runtime assertion checking might differ from that of static verification.

Clearly, mappings between model classes and built-in theories should not be merely based on the names of operations, as pointed out by Guttag [51]:

> *To rely on one's intuition about the meaning of names can be dangerous even when dealing with familiar types.*

A technique that maps model classes to theories of theorem provers must provide a systematic approach, which guarantees that semantic mismatches are prevented. Coming up with such an approach might be tricky when, for instance, the universes or provided operations of related structures do not match.

Contributions. We propose an approach for the faithful mapping of model classes to mathematical structures. Faithfulness means that a given model class semantically corresponds to the mathematical structure it is mapped to. Our approach takes the specifications of model classes into account and poses proof obligations to ensure semantic correspondence of related entities.

The approach enables sound reasoning about programs specified in terms of model classes and helps in improving the quality of model class specifications by making them consistent and complete.

We discuss the consequences of imperfect mappings on the verification process, that is, the consequences of mappings that relate model classes with structures whose sets of operations do not exactly match.

1.3 Outline

Chapter 2 introduces JML^{--}, the language we consider in the thesis, the logic and store model that is used throughout the thesis, and a number of definitions and theorems of mathematical logic that are referred to in the thesis. That preliminary chapter is followed by two parts.

Part I presents the techniques we developed for the encoding of specification expressions, for the checking of their well-definedness, and for the sound axiomatization of pure methods.

Chapter 3 motivates the need for and describes the difficulties of the three techniques through examples. Then, through Chapters 4 to 6, the details of the three techniques are developed. Chapter 7 describes our implementation of the techniques in the Spec# verification system.

Part II presents the technique that we developed for the faithful mapping of model classes to mathematical structures provided by theory libraries of theorem provers.

Chapter 8 motivates our work and introduces a model class and a structure that are used for demonstration purposes in latter chapters. Chapter 9 describes the formal details of our approach, Chapter 10 discusses different aspects of the proposed technique, and Chapter 11 reports on a case study. Chapter 12 extends the technique to the handling of inductively defined data types. Part II is closed by a summary of related work in Chapter 13.

Finally, in Chapter 14 we conclude the thesis by summarizing the main contributions and pointing out areas of further work.

Chapter 2

Preliminaries

2.1 Syntax and Semantics of JML^{--}

In this section, we define the concrete syntax and the semantics of JML^{--}, the language that we consider in this thesis. JML^{--} is essentially a subset of JML (which is a superset of Java), however, it contains a few additional elements introduced for the purposes of the thesis.

JML^{--} contains the most important object-oriented features, such as subtyping, inheritance, and dynamic method binding; and the most important features of interface specifications, namely, object invariants and method specifications.

JML^{--} is minimal in the sense that it contains only those constructs that are needed for the purpose of the thesis. For instance, the language does not contain visibility modifiers because these modifiers play no role in the techniques presented in the sequel. During the introduction of the grammar, we will point out the most important deviations from Java and JML.

The syntax and semantics of the language elements borrowed from Java are left unchanged. We assume basic knowledge of Java, and thus do not go into details of the syntax and semantics of the different constructs and modifiers. The syntax and semantics of JML^{--}'s specification constructs are similar but not identical to that of JML. The precise syntax and semantics of the constructs is given below.

The grammar of JML^{--} is given in Figure 2.1 using the Backus-Naur Form (BNF). Terminals are (1) keywords such as **`class`** and **`new`**, typeset in **`bold typewriter`** font, (2) symbols such as `,` and `==>`, typeset in `typewriter` font, and (3) identifiers such as ClassId and FieldId, typeset in sans-serif font. Nonterminals are typeset in *italic* font. As common in BNF, $a^?$ means a occurring zero or one time, a^+ means a occurring one or more times, and a^* means a occurring zero or more times. Parentheses in roman font "(...)" are used for grouping.

Program	::=	*Type-decl*$^+$
Type-decl	::=	*Type-map** *Modifier** (*Class-decl* | *Iface-decl*) `{` *Member** `}`
Type-map	::=	**`mapped_to`**`("`*String*`","`*String*`","`*String*`");`
Modifier	::=	**`abstract`** | **`pure`** | **`model`** | **`immutable`** | **`helper`**
	|	**`resultNotNewlyAllocated`** | **`noReferenceComparison`**
	|	**`constructing`**
Class-decl	::=	**`class`** ClassId (**`extends`** ClassId)$^?$ (**`implements`** *IfaceId-list*)$^?$
Iface-decl	::=	**`interface`** IfaceId (**`extends`** *IfaceId-list*)$^?$
IfaceId-list	::=	IfaceId (`,` IfaceId)*
Member	::=	*Field-decl* | *Method-decl* | *Invar-decl*
Field-decl	::=	*Type* FieldId`;`
Method-decl	::=	*Method-sign Method-spec*$^?$ (*Method-body* | `;`)
Method-sign	::=	*Method-map** *Modifier** *Type*$^?$ MethodId`(` *Param-list*$^?$ `)`
Method-map	::=	**`mapped_to`**`("`*String*`","`*String*`");`
Param-list	::=	*Param-decl* (`,` *Param-decl*)*
Param-decl	::=	*Type* ParamId
Invar-decl	::=	**`redundantly`**$^?$ **`invariant`** *Expr* `;`
Method-spec	::=	**`redundantly`**$^?$ (**`requires`** *Expr* `;` | **`ensures`** *Expr* `;`)$^+$
		(**`measured_by`** *Expr* `;`)$^?$
Expr	::=	*Expr* ⊛ *Expr*
	|	`!` *Expr*
	|	*Expr* `.` FieldId
	|	*Expr* `.` MethodId `(` *Expr-list*$^?$ `)`
	|	**`new`** ClassId `(` *Expr-list*$^?$ `)`
	|	ParamId
	|	*Literal*
	|	`\`**`old`** `(` *Expr* `)`
	|	`\`**`fresh`** `(` *Expr* `)`
	|	*Quantification*
⊛	::=	`&&` | `||` | `==>` | `==` | `!=` | `<` | `<=` | `+` | `-` | `*` | `/` | `%`
Literal	::=	**`this`** | **`null`** | `\`**`result`** | **`true`** | **`false`** | `-1` | `0` | `1` | ...
Expr-list	::=	*Expr* (`,` *Expr*)*
Quantification	::=	`(` *Quantor* *BVar-decl* `.` *Expr* `)`
Quantor	::=	`\`**`forall`** | `\`**`exists`**
BVar-decl	::=	*Type* BVarId
Type	::=	**`int`** | **`boolean`** | **`void`** | **`nullable`**$^?$ *Ref-type*
Ref-type	::=	ClassId | IfaceId
String	::=	string of characters

Figure 2.1: Grammar of JML^{--}

Programs. A JML⁻⁻ *program* consists of a number of *type declarations*. A type is either a *class* or an *interface*. We assume a set of identifiers for classes and interfaces, ClassId and IfaceId, respectively. Types may be mapped to abstract data types using **`mapped_to`** clauses. The details on the use and semantics of the clause are given in Part II.

A class may be declared to be a subtype of at most one other class and of any number of interfaces. An interface may be declared to be a subtype of any number of other interfaces. The subtype relation must be acyclic.

A class can be declared to be *abstract*. A class or interface can be declared to be *pure*, meaning that all its methods are pure, that is, free of side-effects. A class or interface can be declared to be a *model* type, meaning that the class or interface is only used for specification purposes. A type may be declared to be *immutable*, meaning that the state of an instance of the type may not be altered after initialization of the instance. Other modifiers must not be used for types.

Remark. Type declarations in JML⁻⁻ have no visibility modifiers (such as **`public`**), and types may not be organized into packages. We assume that there is one package in which every type resides and types are mutually visible to each other. These restrictions are made for the sake of simplicity since the techniques proposed in the thesis are orthogonal to the concepts of visibility and packages. An extension to include these concepts is straightforward.

Members. Classes and interfaces have an arbitrary number of *field declarations*, *method declarations*, and *invariant declarations*.

A field declaration consists of a *type* and an identifier from FieldId. A *method declaration* declares a method or a constructor, and consists of a *method signature*, an optional *method specification*, and a possibly missing *method body*.

A method signature consists of an optional *mapping*, the return type, an identifier from MethodId, and a possibly empty list of *formal parameters*. For constructors, no return type is declared.

A method or constructor of a model type may be mapped to a function symbol of some theory specified by **`mapped_to`** clauses, and may be declared to be *constructing*. The details on the use and semantics of these constructs are given in Part II. A method may be declared to be *abstract*. A method or constructor may be declared to be *helper*, meaning that declared invariants do not apply to the method or constructor (see below).

Remark. For simplicity, JML⁻⁻ does not support static methods. An extension to include static methods is mostly straightforward since their encoding and axiomatization is analogous to that of non-static methods.

Remark. In most aspects that are relevant for this thesis, constructors and methods are analogous. Therefore, from now on, we will refer to both methods and constructors by *methods*, and will explicitly distinguish between the two only if necessary.

A method may be declared to be *pure*, which marks it to be side-effect free. Due to their side-effect freeness, pure methods may be invoked in specification expressions. Throughout the thesis, we use the following definition for purity:

Definition 2.1. ***(Purity)*** *A method is pure if it does not alter the state of objects that existed before its execution. The same applies for constructors, except that they are allowed to modify the state of the object that is being initialized.*

Note that this definition allows pure methods to allocate and initialize new objects. This notion of purity is commonly referred to as *weak purity* [11, 103].

Modifier `resultNotNewlyAllocated` for a reference-type pure method means that the returned object is not newly allocated, that is, it was already allocated in the prestate of the method. Constructors may not be marked with this attribute. Modifier `noReferenceComparison` for a pure method means that the specification and implementation of the method does not use operators `==` and `!=` on operands of reference type. The purpose of these modifiers and the way their semantics is enforced is explained in Chapter 4.

Based on the JML Reference Manual [77], we make the assumptions that pure methods terminate and that they are deterministic.

Assumption 2.1. ***(Pure methods terminate)*** *Pure methods and constructors always terminate. They either terminate normally or throw an exception.*

Regarding determinism, we make different assumptions based on the return values of pure methods:

Assumption 2.2. ***(Pure methods are deterministic)*** *Pure methods and constructors are deterministic. (1) A pure method with primitive return-type returns the same value when called in a given state. (2) A pure method that is marked with modifier* `resultNotNewlyAllocated` *returns objects that are equal in terms of reference equality when called in a given state. (3) A pure constructor or a method not marked with that modifier returns objects that are equal in terms of the* `equals` *method when called in a given state.*

This means that methods whose behavior depend, for instance, on some random-number generator or the system time, cannot be declared as pure. However, we allow, for instance, method `hashCode` declared in class `Object` to be marked as pure because in a given state it returns the same "address".

The fact that two consecutive executions of the same expression (for example, `new C().hashCode()`) yield different results does not mean that the method is non-deterministic because the method is called in two different states.

Remark. Although languages may provide means other than just `hashCode` to get hold of values that are related to object identities,[1] in the sequel, we will only mention `hashCode` to refer to these means.

It is out of the scope of this thesis to develop techniques that would justify that these assumptions are valid for given pure-method implementations. Here we just briefly refer the reader to approaches that can be applied to justify these assumptions.

Termination is typically proven by the use of logics that ensure *total correctness*. Such logics not only show that *if* the program terminates then its behavior is correct (partial correctness), but also that it *does* terminate. Such logics are developed in standard textbooks, for instance, in that of Nielson and Nielson [105].

A technique capable to determine whether a pure method is deterministic or not is that of equivalent-results methods by Leino and Müller [85]. The technique uses self-composition to simulate two executions of the method body. Results of the two "runs" are compared against a user-defined equivalence relation to see if the method is deterministic modulo the defined relation.

Note that the nonterminal *Method-body* is not defined. This is because throughout the thesis we will not be concerned with implementations, and thus the syntax of statements is irrelevant. In the sequel, examples with implementations will use Java's statement-syntax.

Remark. Method and constructor declarations in JML⁻⁻ have no visibility modifiers. Again, the concept of visibility is orthogonal to the topic of the thesis and an extension to include it is straightforward. Throughout the thesis we consider every method to be visible for every type.

Interface Specifications. We consider two kinds of interface specifications: *object invariants* and *method specifications*. The invariant of an object is the conjunction of expressions specified in invariant declarations. If no invariant is declared then the object invariant defaults to **`true`**. An object invariant specifies the consistent states of objects that are instances of the enclosing type. The literature defines different invariant semantics, that is, different definitions when and which invariants must hold during program

[1] The `hashCode` method defined by class `Object` returns distinct integers for distinct objects; typically by converting internal addresses into integers [66].

execution [100, 9, 115, 102]. The invariant semantics used in the thesis is the *visible-state semantics*, which is the invariant semantics used by JML [77]. In order to give its definition, we first define *visible states* based on Müller *et al.* [102]:

Definition 2.2. ***(Visible states)*** *A program execution state is called visible if it is the prestate or the poststate of a non-helper method execution. The* prestate *of a method execution is the state immediately before the execution of the method body, that is, after the actual parameter values of the call have been assigned to the formal parameters of the method and control has been transferred to the method. The* poststate *of a method execution is the state immediately after the execution of the method body before control is transferred back to the caller.*

The visible-state semantic is defined as follows [77]:

Definition 2.3. ***(Visible-state semantics)*** *The object invariants of all allocated objects must hold in all visible states of a program, except in the prestate of constructors where the invariant of the object that is being initialized need not hold.*

A method specification prescribes the expected functional behavior of a method. A method specification consists of at least one *requires* or *ensures* clause. Requires clauses specify the precondition of the method at hand, and ensures clauses the postcondition. The *measured_by* clause is used in termination arguments for recursively specified methods. The argument of the clause has to be of type `int`.

If a method has no method specification or its method specification does not contain a requires clause, then the precondition of the method defaults to `true`. If multiple requires clauses are given in a method specification, then the precondition of the method is the conjunction of the expressions in the clauses. The rules are analogous for ensures clauses.

The semantics of pre- and postconditions is the following:

Definition 2.4. ***(Semantics of pre- and postconditions)*** *If a method is called in a prestate where all required object invariants and the method's precondition hold, then the method must terminate normally in a poststate where all required object invariants and the method's postcondition hold.*

Remark. The above semantics requires normal termination, that is, methods may not diverge and may not throw exceptions. In JML, this type of method specification is called *normal behavior specification case*. JML^{--} only supports this kind of specification case. As explained in Section 6.1, method behavior that may throw an exception is orthogonal to the techniques described in the thesis. Therefore, an extension to other specification cases is straightforward.

Remark. JML^{--} allows one to specify a given method only by one specification case, that is, only by one pre- and postcondition pair (via a set of requires and ensures clauses). This restriction is made to simplify formalizations throughout the thesis, in particular, in Chapter 6. We refer the reader to [112] for a detailed account on how multiple method specification cases can be desugared into a single one while retaining semantical equivalence.

Remark. JML^{--} specifications do not contain means to specify frame properties (specified by **`assignable`** clauses in JML). This is because the focus of the thesis is on pure methods. As expressed by Definition 2.1, pure methods have an implicit frame property expressing that they may not modify pre-existing state (specified by the implicit **`assignable \nothing`** clause for such pure methods in JML). An extension to include frame properties is straightforward.

An invariant or method specification may be declared to be *redundant*, meaning that the property expressed by the specification is derivable from other, non-redundant specifications.

Behavioral Subtyping. In the realm of verification of object-oriented programs, a crucial property is behavioral subtyping defined by Liskov and Wing [89]:

Definition 2.5. ***(Behavioral subtyping)*** *Let $\phi(o)$ be a property about objects o of type T such that $\phi(o)$ is provable by the specification of type T. Then $\phi(o')$ should be true for objects o' of type S, where S is a subtype of T.*

Behavioral subtyping can be enforced in different ways. The most commonly used techniques are the ones proposed by Liskov and Wing [89], and Dhara and Leavens [39]. The former requires a certain logical relation between the specifications of supertypes and subtypes. The latter achieves these logical relations by defining a specification semantics where subtypes inherit the specifications of supertypes. The semantics of JML prescribes the latter approach [39].

We do not prescribe how behavioral subtyping is guaranteed, but simply make the assumption that:

Assumption 2.3. *Subtypes are behavioral subtypes.*

Expressions. Expressions may contain the binary operators listed under $\circledast$. All operators except `==>` are present in Java, and their semantics is also the same as in Java. The operator `==>` denotes "short-circuit" logical implication.

Expressions may contain the unary logical negation.

Expressions may contain field reads and calls to pure methods. Expressions may also contain formal parameters as well as the literals **this**, **null**, **true**, **false**, and integer literals.

Postconditions may contain three constructs that are not part of Java, but are typically part of modern specification languages. The **old** construct yields the value of its argument in the prestate of the specified method. The keyword **result** refers to the return value of the non-void method being specified. In constructors, **this** refers to the object being initialized. The **fresh** construct expresses that its reference-type argument gets allocated by the specified method. That is, the object referenced by the argument was not allocated in the prestate and is non-null in the poststate of the method.

Following the semantics of JML, if a formal parameter appears in a postcondition then it denotes the value of the parameter in the prestate—even if the parameter occurs outside an old-expression. For instance, if `p` is a formal parameter then `p` and **old**`(p)` denote the same value in every pre- and poststates. However, note that `p.f` and **old**`(p.f)` may denote different values as location `p.f` (or **old**`(p).f`) may have different values in given pre- and poststates.

In contrast to formal parameters, **result** is always bound to the poststate, even if it occurs in an old-expression. In order to rule out expressions that do not denote any value (such as **old**`(`**result**`.f)` if **result** denotes a newly-allocated object), we make the following restriction: Specifications of methods that are not marked with modifier **resultNotNewlyAllocated** must not mention keyword **result** in old-expressions.

As we will see in Section 4.1, our encoding of constructors unifies the (semantically separate) steps of allocating and initializing the new object. Due to this encoding, the state after the allocation and before the initialization is not observable by our approach. Thus, keyword **this** in the specification of constructors is analogous to keyword **result** in the specification of methods. Therefore, in the specification of constructors we forbid keyword **this** to be mentioned in preconditions and old-expressions. Forbidding such specifications does not seem to cause any limitation in practice.

Finally, expressions may contain universal and existential quantification over variables of integer, boolean, and reference type. Quantification over variables of reference type ranges over allocated objects of the corresponding type. Due to the way constructors are encoded, the range excludes the object denote by **this** in the prestate of constructors.

Remark. The semantics defined above for quantifiers differs from the semantics defined by JML. In JML, quantification over variables of reference type ranges over all objects, even *non-allocated* ones. The reason for this deviation in the semantics will be explained in latter chapters.

Types. JML^{--} has two primitive types: **`int`** and **`boolean`**. Reference types represent references to objects. Following the semantics of JML, by default, declared fields, method parameters, and return values of reference type may not be **`null`**. To allow them to take the **`null`** value, one has to use the **`nullable`** modifier in field declarations and method signatures.

2.2 Logic and Store Model

In this section, we describe the logic and the model of the object store that is used throughout the thesis.

Logic. The logic that is used in Part I is two-valued first-order logic. The choice of first-order logic is a direct consequence of the specification language being first order. Two-valued logic was chosen because it is simpler and better understood than many-valued logic, moreover, two-valued logic is more widely used and has better automated tool support than many-valued logic [55].

Part II uses two-valued higher-order logic. Although the presented technique is not bound to any particular logic, the provided examples and case studies map model classes to Isabelle/HOL theories.

Store Model. To formalize properties of the object store, we use the store model of Poetzsch-Heffter and Müller's program logic [111]. It is formalized in multi-sorted first-order logic with recursive data types.

Types and Values. Java's types and values are modeled by the sorts **Type** and **Value**, respectively. Sort **Type** contains primitive types, the type of the **`null`** reference, and class types. The reflexive, transitive subtype relation is denoted by $\preceq$. A **Value** is a value of a primitive type, the **`null`** reference, or a reference to an object. The function *typeof* : **Value** $\rightarrow$ **Type** yields the type of a value.

Object States. Object states are modeled via *locations* (instance variables). For each field of its class, an object has a location. The sort FieldId is the sort of unique field identifiers of a program. The function $loc(X, f)$ either yields the location for field f of the object referenced by X or *undefined*, denoted by $\perp$, if the object does not have a location for f. Conversely, $obj(L)$ yields a reference to the object a location L belongs to. For brevity, we will often write $X.f$ for $loc(X, f)$ in the following.

$$\begin{aligned} loc &: \textbf{Value} \times \mathsf{FieldId} \rightarrow \textbf{Location} \cup \{\perp\} \\ obj &: \textbf{Location} \rightarrow \textbf{Value} \end{aligned}$$

Since the properties of these functions are not needed in this paper, we refer the reader to [111] for their axiomatization.

Object Stores. Object stores are modeled by an abstract data type with main sort **Store** and operations to read and update locations, to create new objects, and to test whether an object is allocated. Poetzsch-Heffter and Müller present these functions and their axiomatization [111].

In this thesis, we need the following store operations: $OS\langle T\rangle$ yields the object store that is obtained from OS by allocating a new object of class T. $new(OS, T)$ yields a reference to this object. The lookup function $OS(L)$ denotes the value held by location L in store OS. $alive(X, OS)$ yields true if and only if (1) X is of primitive type, (2) X is the `null` reference, or (3) X denotes an object that is allocated in OS. The sort ClassId is the sort of unique class identifiers of a program.

$$\begin{array}{lcl} \_\langle\_\rangle & : & \textbf{Store} \times \mathsf{ClassId} \rightarrow \textbf{Store} \\ new & : & \textbf{Store} \times \mathsf{ClassId} \rightarrow \textbf{Value} \\ \_(\_) & : & \textbf{Store} \times \textbf{Location} \rightarrow \textbf{Value} \\ alive & : & \textbf{Value} \times \textbf{Store} \rightarrow \textbf{Bool} \end{array}$$

The logic uses the special program variable $resV$ to represent the return value of a non-void method.

In order to make formulas more readable, we introduce two additional functions with the following signatures and definitions:

$$\begin{array}{lcl} alloc & : & \textbf{Value} \times \textbf{Store} \rightarrow \textbf{Bool} \\ allocT & : & \textbf{Value} \times \textbf{Store} \times \textbf{Type} \rightarrow \textbf{Bool} \end{array}$$

$$\begin{array}{lcl} alloc(o, OS) & \triangleq & typeof(o) \preceq Object \ \wedge \ alive(o, OS) \ \wedge \ o \neq null \\ allocT(o, OS, T) & \triangleq & alloc(o, OS) \ \wedge \ typeof(o) \preceq T \end{array}$$

We will say that "an object o is *allocated* in store OS" to mean that $alloc(o, OS)$ holds.

Remark. While *allocatedness* is only meaningful when talking about objects, *aliveness* is meaningful whenever talking about values. For instance, according to the definitions of *alive* and *alloc*, `null` and 5 are always alive but never allocated.

Remark. Once an object is allocated in store OS, it remains alive (and therefore also allocated) in every successor store of OS. Thereby, the program logic is not sensitive to garbage collection.

2.3 Bit of Mathematical Logic

This section summarizes standard notions of mathematical logic that will be used in the thesis. We assume basic knowledge of these notions and results,

and do not go into explanations. Interested readers are referred to standard textbooks, such as [125, 41].

Syntax. The syntax of first-order logic is defined as follows. The language signature Σ is defined by a set $\mathbf{V}$ of variable symbols, a set $\mathbf{F}$ of function symbols, and a set $\mathbf{P}$ of predicate symbols. First-order terms and formulas over Σ are defined by the following syntax:

$$
\begin{array}{rcl@{\qquad}rcl}
\mathit{Term} & ::= & \mathit{Var} & \mathit{Formula} & ::= & P(t_1, \ldots, t_n) \\
& \mid & f(t_1, \ldots, t_n) & & \mid & \mathit{true} \quad \mid \quad \mathit{false} \\
& & & & \mid & \neg\phi \quad \mid \quad \phi_1 \Rightarrow \phi_2 \\
& & & & \mid & \phi_1 \wedge \phi_2 \quad \mid \quad \phi_1 \vee \phi_2 \\
& & & & \mid & \forall\, x.\ \phi \quad \mid \quad \exists\, x.\ \phi
\end{array}
$$

where $\mathit{Var} \in \mathbf{V}$, $f \in \mathbf{F}$, $P \in \mathbf{P}$, t_i are terms, and ϕ, ϕ_i are formulas.

The first-order language L' is an *extension* of the first-order language L if every symbol in the signature of L is a symbol in the signature of L'.

Structures. Let $\mathbf{A}$ be a non-empty set that does not contain $\bot$. We define $\mathbf{A}_\bot$ as $\mathbf{A} \cup \{\bot\}$. Let $\mathbf{I}$ be a mapping from $\mathbf{F}$ to the set of functions from $\mathbf{A}^n$ to $\mathbf{A}_\bot$, and from $\mathbf{P}$ to the set of predicates from $\mathbf{A}^n$ to $\mathbf{Bool}_3 \triangleq \{\mathbf{true}, \mathbf{false}, \bot\}$, where n is the arity of the corresponding function or predicate symbol.

We say that $\mathbf{M} = \langle \mathbf{A}, \mathbf{I} \rangle$ is a *structure* for our language with *carrier set* $\mathbf{A}$ and *interpretation* $\mathbf{I}$. We call a structure *total* if the interpretation of every function $f \in \mathbf{F}$ and predicate $P \in \mathbf{P}$ is total, which means $f(\ldots) \neq \bot$ and $P(\ldots) \neq \bot$. We call the structure *partial* otherwise. A partial structure $\mathbf{M}$ can be *extended* to a total structure $\widehat{\mathbf{M}}$ by having functions evaluated outside their domains yield arbitrary values.

Interpretation. For some total structure $\mathbf{M}$ and variable assignment θ, we denote the interpretation of term $\mathbf{t}$ in two-valued logic as $[\mathbf{t}]^2_{\mathbf{M}}\theta$, and the interpretation of formula φ as $[\varphi]^2_{\mathbf{M}}\theta$. *Variable assignment* θ maps the free variables of $\mathbf{t}$ and φ to values. The two-valued interpretation is the standard one [125, 41].

For some structure $\mathbf{M}$ and variable assignment θ, we denote the interpretation of term $\mathbf{t}$ and formula φ in three-valued logic as $[\mathbf{t}]^3_{\mathbf{M}}\theta$ and $[\varphi]^3_{\mathbf{M}}\theta$, respectively. The three-valued interpretation of terms and formulas can be defined in different ways. For instance, logical connectives can be interpreted according to Kleene's or McCarthy's semantics, and the interpretation of functions and predicates may or may not be "strict".[2]

[2] In a strict interpretation, functions and predicates yield $\bot$ in case any of their arguments is interpreted as $\bot$.

The actual three-valued interpretation of terms and formulas used in the thesis will be given later.

Models. The *truth* of formula φ for structure $\mathbf{M}$ and variable assignment θ is denoted by $(\mathbf{M}, \theta) \models \varphi$. Structure $\mathbf{M}$ is a *model of formula* φ, denoted as $\mathbf{M} \models \varphi$, if $(\mathbf{M}, \theta) \models \varphi$ for all variable assignments θ. A formula φ is *valid*, denoted as $\models \varphi$, if for every structure $\mathbf{M}$ and variable assignment θ: $(\mathbf{M}, \theta) \models \varphi$. A *theory* T consists of a set of formulas that contain no free variables: $T = \{\varphi_1, \ldots, \varphi_n\}$. We will often refer to these formulas as *axioms*. A formula φ is a *theorem of theory* T if φ is a consequence of the axioms of T. A theory T is called *inconsistent* if every formula ψ is a theorem of T. A theory is called *consistent* otherwise. Structure $\mathbf{M}$ is a *model of theory* T, denoted as $\mathbf{M} \models T$, if $\mathbf{M} \models \varphi_i$ for all $\varphi_i \in T$. An important property for our purposes is the following theorem.

Theorem 2.1. *A theory T is consistent if and only if it has a model.*

Remark. In the definitions above, it was not specified, for instance, whether structure $\mathbf{M}$ was partial or total, and whether two- or three-valued interpretation was meant. This means that we will "overload" notations, however the context in which they appear will make the exact meaning clear.

A theory T' is an *extension* of a theory T if the language of T' is an extension of the language of T and every theorem of T is a theorem of T'. A *conservative extension* of T is an extension T' of T such that every formula of T that is a theorem of T' is also a theorem of T.

Well-definedness. For total structures, the interpretation maps a formula to a value in $\mathbf{Bool} \triangleq \{\mathbf{true}, \mathbf{false}\}$. For partial structures, the interpretation maps a formula to a value in $\mathbf{Bool}_3 \triangleq \{\mathbf{true}, \mathbf{false}, \bot\}$. To check whether or not a value in $\mathbf{Bool}_3$ is $\bot$, we use function $\mathbf{wd}$:

$$\mathbf{wd} : \mathbf{Bool}_3 \to \mathbf{Bool}$$

$$\mathbf{wd}(x) \triangleq \begin{cases} \mathbf{true} & , \text{ if } x \in \{\mathbf{true}, \mathbf{false}\} \\ \mathbf{false} & , \text{ if } x = \bot \end{cases}$$

Theory Interpretation. Different authors present the notion of *theory interpretation* differently. Below, we follow the presentation of Farmer [42].

An n-ary *term function* is a λ-expression $\lambda\{x_1, \ldots, x_n.\ E\}$, where E is a term. Analogously, if E is a formula, then the λ-expression is called an n-ary *formula function*.

Let T and T' be two first-order theories. A *standard translation from T to T'* is a pair (U, ν) where U is a closed unary predicate and ν is a function from the nonlogical symbols of T to the nonlogical symbols and terms of T' such that:

1. if f is an n-ary function symbol of T, then $\nu(f)$ is either an n-ary function symbol or a closed n-ary term function;
2. if P is an n-ary predicate symbol of T other than $=$, then $\nu(P)$ is either an n-ary predicate symbol or a closed n-ary formula function;
3. $\nu(=) \triangleq =$.

Let $\Phi = (U, \nu)$ be a standard translation from T to T'. For a term [formula] E of T, the *translation of E via Φ*, written $\Phi(E)$, is the term [formula, respectively] of T' defined inductively as follows:

1. $\Phi(x) \triangleq x$, if x is a variable;
2. $\Phi(S(t_1, \ldots, t_n)) \triangleq \nu(S)(\Phi(t_1), \ldots, \Phi(t_n))$, if S is an n-ary function or predicate symbol;
3. $\Phi(\mathit{true}) \triangleq \mathit{true}$ and $\Phi(\mathit{false}) \triangleq \mathit{false}$;
4. $\Phi(\neg\phi) \triangleq \neg\Phi(\phi)$;
5. $\Phi(\phi \circ \psi) \triangleq \Phi(\phi) \circ \Phi(\psi)$, if $\circ \in \{\wedge, \vee, \Rightarrow\}$;
6. $\Phi(\forall x.\ \phi) \triangleq \forall x.\ U(x) \Rightarrow \Phi(\phi)$;
7. $\Phi(\exists x.\ \phi) \triangleq \exists x.\ U(x) \wedge \Phi(\phi)$.

A standard translation associates the universe of its source theory with a closed unary predicate of the target theory. In the sequel, we will refer to this predicate as the *universe predicate*. The translation associates the nonlogical symbols of the source theory with closed terms and formulas of the target theory, and variables and logical connectives with themselves. The quantifiers are *relativized* to the universe predicate.

Φ is a *standard interpretation of T in T'* if $\Phi(\phi)$ is a theorem of T' for each theorem ϕ of T. A sufficient condition for a standard translation Φ from T to T' to be a standard interpretation is that the following formulas, called *obligations*, are theorems of T':

1. $\Phi(\phi)$ for each axiom ϕ of T; *(axiom obligation)*
2. $\exists x.\ U(x)$; *(universe nonemptiness obligation)*
3. $\forall x_1, \ldots, x_n.\ U(x_1) \Rightarrow \ldots \Rightarrow U(x_n) \Rightarrow U(\Phi(f(x_1, \ldots, x_n)))$ for each function symbol f of T.[3] *(function symbol obligation)*

The intuition behind the axiom and universe nonemptiness obligations is straightforward. The function symbol obligation expresses that the interpretation of a function f is a function whose restriction to the universe takes values in the universe.

[3] Farmer [42] presents the property in a different, less intuitive form. Thus, the form used by Shoenfield [125] is used instead.

If the universe of the source and the target theory is identical, then $U(x) \triangleq (x = x)$. Consequently, (1) the relativization of quantifiers can be omitted in translations and (2) the second and third obligations trivially hold.

An important property for our purposes is the following theorem.

Theorem 2.2. *If there is a standard interpretation of T in T', and T' is consistent, then T is consistent.*

The following example, presented by Farmer [42], shows a simple example of a standard first-order interpretation.

Example 2.1. Let T be the theory consisting of the following three axioms of a binary relation symbol $\leq$:

1. Reflexivity: $\forall x.\ x \leq x$
2. Transitivity: $\forall x, y, z.\ x \leq y\ \wedge\ y \leq z\ \Rightarrow\ x \leq z$
3. Antisymmetry: $\forall x, y.\ x \leq y\ \wedge\ y \leq x\ \Rightarrow\ x = y$

T specifies $\leq$ to be a nonstrict partial order.

Let T' be the theory consisting of the following three axioms of a binary relation symbol $<$:

1. Irreflexivity: $\forall x.\ \neg(x < x)$
2. Transitivity: $\forall x, y, z.\ x < y\ \wedge\ y < z\ \Rightarrow\ x < z$
3. Trichotomy: $\forall x, y.\ x < y\ \vee\ y < x\ \vee\ x = y$

T' specifies $<$ to be a strict total order.

Let $\Phi = (U, \nu)$ be the standard translation from T to T' where

$$U(x) = (x = x) \quad \text{and} \quad \nu(\leq) = \lambda\{x, y.\ x < y\ \vee\ x = y\}$$

It can be easily proven that Φ is a standard interpretation: the axiom obligation can be shown by a few proof steps and, due to the defined universe predicate, the second and third obligations trivially hold. □

Part I

Encoding and Axiomatization of Pure Methods

Chapter 3

Motivation

The use of one-tiered specification languages allows programmers to write specifications in a syntax that is nearly identical to the one they use for writing programs. For instance, the syntax of JML^{--} expressions allows specifiers to use the logical operator `&&`, to refer to the **`this`**-object, to access fields of objects, and to call pure methods.

This flexibility comes at the price that the semantic gap between the constructs of JML^{--} expressions and first-order logic has to be bridged in order to reason about specifications. In Part I of the thesis we show three means that provide a solution for filling this semantic gap: (1) an *encoding* of specification expressions in first-order logic, (2) a technique that ensures *well-definedness* of specification expressions by posing proof obligations, and (3) a technique for the sound *axiomatization* of pure methods.

In this chapter, we motivate the need for and the difficulties of these techniques through examples. Furthermore, we explain the relation between the three techniques.

3.1 Encoding of Specification Expressions

The static verification of programs specified with a one-tiered specification language requires an encoding of specification expressions in the underlying logic of the verification environment at hand. Such an encoding is indispensable as the domain of the specification language is considerably different from that of the logic. For instance, the expression `\fresh(this.f)` does not have a meaning in first-order logic.

One of the typical usages of such an encoding is the generation of proof obligations for verifying the correctness of implementations. For instance, our previous work describes the way Hoare-triples are generated for a subset of JML in the Jive verification system [34]. In this thesis, the encoding will serve two other purposes: checking the well-definedness of specification expressions and extracting axioms from the specifications of pure methods.

Next, we show the issues that the design of such an encoding faces.

3.1.1 Preserving Semantics of Expressions

As mentioned earlier, a crucial requirement for an encoding is that it preserves the original semantics of specification expressions. However, such an encoding is non-trivial. For instance, JML^{--} expressions implicitly refer to state (*e.g.*, via field accesses, method calls, and the `\fresh` construct), while state is not a first-class citizen in logic and thus has to be modeled somehow.

The encoding that we propose in Chapter 4 interprets JML^{--} constructs in terms of the store-operations that were presented in Section 2.2. These operations provide a rather straightforward encoding of most store-related expressions. The only exception is the case of method calls. Calls are not covered by the store-formalization of Poetzsch-Heffter and Müller [111] because it was developed in a two-tiered setting. The main issues raised by calls in specification expressions are discussed in the next section.

Another difference between the semantics of JML^{--} expressions and classical two-valued logic is the interpretation of logical operators. In particular, the interpretation of "short-circuit" operators, as illustrated by the following example.

Example 3.1. Assume a method with two integer parameters `p1` and `p2`, and the precondition "`p1/p2 > 0 || true`". If an encoding would simply translate operator `||` to logical disjunction, then the resulting formula would be "$div(p1, p2) > 0 \vee true$", where function *div* denotes division in the underlying logic. However, this encoding might not preserve the original semantics: On the one hand, due to the left-to-right evaluation, the expression does not denote a value if `p2` is 0; on the other hand, classical two-valued logic does not have a notion of evaluation order, and thus the formula always evaluates to **true**. □

Our solution to such discrepancies, presented in Chapter 5, is a requirement that every specification expression must be well-defined. Under this requirement, we get a matching semantics among JML^{--} operators and logical connectives that the encoding relates. The challenges of actually enforcing the requirement are discussed in Section 3.2.

3.1.2 Object Allocation

Allowing pure methods to allocate and initialize new objects is important for expressiveness [103, 118]. Object-oriented languages represent almost all data as objects. Therefore, methods that return strings, tuples, sequences, sets, etc. often create objects. Moreover, methods often create and manipulate auxiliary objects, for instance, iterators. Such methods do not modify objects that exist in the prestate, but they are not entirely free from side-effects.

The following three examples demonstrate that ignoring the side-effects possibly made by pure methods in the encoding may lead to unexpected results during verification.

Example 3.2. Class `Unsound` in Figure 3.1 has a field `f` and an invariant that requires `f` to be non-zero. `Unsound`'s constructor is declared to be pure and helper, which allows it to return an object that does not satisfy its invariant. In fact, the `f` field of the new object is initialized to zero, as stated in the constructor's ensures clause. The constructor is pure since it modifies only the new object.

The constructor is called in the requires clause of method `divide`. According to the visible state semantics, one can assume that all objects that are allocated and initialized satisfy their invariants in the prestore of `divide`. If an encoding neglects the side-effects of a pure constructor, then one can conclude that after the constructor call still the invariants of all allocated and initialized objects hold (since the store is assumed to be unchanged) and, therefore, `(new Unsound()).f` evaluates to a non-zero value. By this reasoning, one can conclude that `v` is different from zero, which allows one to verify that `divide` does not terminate abruptly.

On the other hand, one can prove that the precondition of the call `divide(0)` in method `showIt` is satisfied because, by the postcondition of the constructor, `(new Unsound()).f` evaluates to zero. Therefore, method `showIt` verifies although it leads to a runtime exception.

The source of the problem is that the specification extends the set of allocated objects, in particular, with an object that violates the invariant. And, this extension is not reflected in the store argument used for the encoding, thus the violation is unnoticed. Note that the example is not bound to constructors: the same issue would arise with a pure helper method that returned a newly-allocated object initialized to `f` being 0. □

Example 3.3. Class `Alloc` in Figure 3.2 declares a pure method `alloc`, which is used in the specification of method `foo`. If an encoding did not consider the possible state changes made by `alloc`, then `foo`'s ensures clause `alloc()==alloc()` would be encoded in a single state. Thus, a uniform handling of pure methods would encode the clause by an equality with two identical operands (concretely, $\widehat{alloc}(this, OS) = \widehat{alloc}(this, OS)$ when encoded in state OS). Such an equality is trivially provable.

However, `alloc` returns a fresh object, as expressed by its postcondition. That is, the returned object does not exist in the prestate of the method. Since the poststate of the first call to `alloc` is the same as the prestate of the second call, the two objects returned by the two calls cannot be equal in terms of reference equality. That is, the postcondition should always evaluate to **false**. In particular, that is the value a runtime assertion checker would yield.

```
class Unsound {
  int f;
  invariant f != 0;

  pure helper Unsound()
    ensures this.f == 0;
  { f = 0; }

  int divide(int v)
    requires v ==
      (new Unsound()).f;
  { return 5 / v; }

  int showIt()
  { return divide(0); }
}
```

Figure 3.1: Pure helper constructor violates invariant of new object

```
class Alloc {

  pure Alloc()
    { ... }

  pure Alloc alloc()
    ensures \fresh(\result);
  { return new Alloc(); }

  void foo()
    ensures alloc() == alloc();
  { ... }
}
```

Figure 3.2: Pure method `alloc` returns a fresh object

The source of the problem is that the faithful modeling of the allocation mechanism requires the encoding to take into account that `alloc` possibly changed the state. Note that the problem would not occur if `alloc` returned a non-fresh object or if reference equality was not tested. □

Example 3.4. Consider the specification of method `alloc` in Figure 3.2. The postcondition expresses that the method's return value is fresh, that is, the value was not allocated in the prestore and is non-null in the poststore. This means that if the pre- and poststore was considered to be identical then the postcondition would always yield **false**. However, in the case of method `alloc` the return value denotes a newly-allocated object, thus the postcondition should always yield **true**.

The source of the problem is that if the return value is a newly-allocated object then that value is an obvious evidence for the pre- and poststore not being identical. □

In Chapter 4, we show how to explicitly model store changes in order to resolve the examples shown above. Furthermore, we show how the modeling can be significantly simplified making it practical for program verification.

3.2 Well-definedness of Specification Expressions

The expression syntax of JML^{--} allows one to apply *partial operators*, such as field access and the division operator. Applications of such operators outside

their domains lead to *ill-defined expressions*. For instance, expressions $o.f$ and x/y are ill-defined if o is `null` and y is 0, respectively.

At runtime, the execution of such expressions typically results in an exception. However, this behavior is usually not desired to be explicitly modeled by an encoding of specification expressions. Therefore, different solutions were proposed for the handling of ill-definedness. For reasons explained in Chapter 5, the solution we adapt in this thesis is the *elimination* of ill-defined expressions. Before the actual verification process, each specification element is checked for well-definedness. Checks are realized by posing proof obligations on the elements.

In the presence of calls to pure methods, determining whether an expression is well-defined or not may require proving arbitrarily complex properties. When a call occurs in an implementation, one has to prove that all object invariants and the precondition of the callee hold. The situation is analogous for calls that occur in specifications. Therefore, to show that a call-expression is well-defined one has to prove that the invariants and the precondition hold.

A further complication is that specification elements may depend on each other. That is, the well-definedness of some specification element may only be provable if some information provided by some other specification element is available. We illustrate such dependencies by the following example [116].

Example 3.5. Consider the abstract class `Sequence`, presented in Figure 3.3. The class contains pure methods to query whether the sequence is empty, and to get the first element and the rest of the sequence. Method `count` returns the number of occurrences of its parameter in the sequence. The class contains field `length`, which represents the length of the sequence. The class contains method specifications and invariants specifying `length`.

Consider the postcondition of method `count`. It contains calls to four different methods, three of which have declared preconditions. For each call, we have to prove, among other things, that the precondition of the callee holds. Let us take a look at expression `rest().count(c)` in the fourth and fifth ensures clauses of method `count`. To show that preconditions are not violated, we need to prove that method `isEmpty` yields **false** both when called on the **`this`**-object and when called on the object that `rest` returns; and we need to prove that `rest` returns a non-null value. The former requirement stems from the declared preconditions of `rest` and `count`, the latter from the implicit precondition of the call to `count` that the receiver must be non-null. The former requirement can be proven by the precondition of `count` and the guarding condition `!rest().isEmpty()`, the latter by the precondition of `count` and the specification of `rest`.

As we can see, in order to prove the well-definedness of the specification of method `count`, we need certain pieces of `count`'s specification as well as the specification of other pure methods. □

```
abstract class Sequence {
  int length;

  invariant length >= 0;
  invariant isEmpty() ==> length == 0;
  invariant !isEmpty() ==> length == rest().length + 1;

  pure abstract int count(Object c)
    requires !isEmpty();
    ensures \result >= 0;
    ensures (!getFirst() == c && rest().isEmpty()) ==> \result == 0;
    ensures ( getFirst() == c && rest().isEmpty()) ==> \result == 1;
    ensures (!getFirst() == c && !rest().isEmpty()) ==>
                                 \result == rest().count(c);
    ensures ( getFirst() == c && !rest().isEmpty()) ==>
                                 \result == rest().count(c) + 1;
    measured_by length;

  pure abstract boolean isEmpty();

  pure abstract Object getFirst()
    requires !isEmpty();

  pure abstract Sequence rest()
    requires !isEmpty();
    ensures \result != null;

  // other methods and specifications omitted
}
```

Figure 3.3: Specification of abstract class `Sequence`

Our solution, presented in Chapters 5 and 6, is to emit well-definedness conditions for each specification element of the program at hand, and to prove these conditions using as much information as possible without the danger of reasoning in an unsound way. To achieve this, we will explicitly track dependencies between specification elements by a *dependency graph*. The traversal of this graph will determine the order in which specification elements are checked and the pieces of information that can be used for proving a given well-definedness condition.

3.3 Axiomatization of Pure Methods

Function symbols that encode pure methods are defined by axioms that reflect the behavior of the methods as expressed by their interface specifications. Generating these axioms is difficult because the axiomatization of the functions has to be consistent to avoid unsound reasoning.

Example 3.6. Consider the following two methods with infeasible specifications:

```
pure abstract int wrong();
  ensures \result == 0;
  ensures \result == 1;
```

```
pure int direct()
  ensures \result == direct() +1;
{ return 5; }
```

The specification of method `wrong` expresses that its return value is 0 and 1. If this property was turned into an axiom, then we could derive that the value of the uninterpreted function introduced for the method is both 0 and 1. From this, we could immediately derive that $0 = 1$, thus **false**.

Method `direct` is specified in terms of itself. The specification is clearly not satisfiable by a pure method, and the recursive call is clearly ill-founded. Again, if this property was turned into an axiom then we could derive the value of the uninterpreted function introduced for the method is the value plus 1. From this, $0 = 1$ and thus **false** follows. □

In practice, unsatisfiable specifications are far less obvious than in the above examples because typically multiple specification elements are involved. The main difficulty of checking feasibility and well-foundedness of specifications lies again in the subtle dependencies between the specification elements, as illustrated by the following example.

Example 3.7. In order to show the feasibility of method `count` of class `Sequence` in Figure 3.3, one has to show that there actually is a result value for each call to method `count`. This would not be the case, for instance, if the first ensures clause required `\result` to be strictly positive because it would contradict the second and possibly the fourth ensures clauses. Since the fourth and fifth ensures clauses of `count` are recursive, proving the existence of a result value relies on the specification of `count`. Using this specification is sound since the recursion in `count`'s specification is well-founded: (1) the first and third invariant, and the precondition of `count` guarantee that the sequence is finite, and (2) the guarding condition together with the precondition of `count` and the third invariant guarantees that we recurse on a shorter sequence. Again, we have a subtle interaction between specifications: proving the consistency of a pure method makes use of the specification of this method as well as invariants and the specification of the methods mentioned in these invariants. □

This example demonstrates that generating the appropriate proof obligations to ensure the feasibility of specifications is non-trivial. In Chapter 6, we present our solution, which considers the dependencies of specification elements and which handles (mutual) recursive specifications.

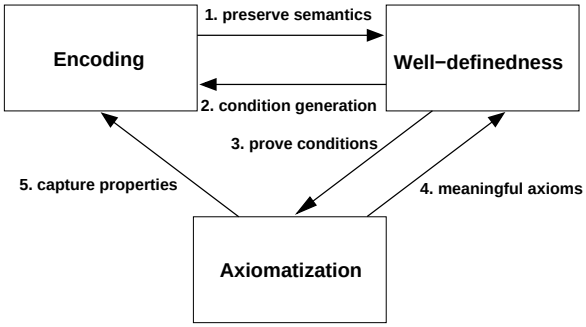

Figure 3.4: Dependencies between the techniques

3.4 Dependencies Between the Techniques

The encoding, well-definedness checking, and axiomatization of specifications are tightly coupled. The dependencies between the techniques are depicted in Figure 3.4, where an arrow from technique A to technique B means that A relies on B. The cyclic dependency between the three techniques indicate that they should be used simultaneously in order to get a semantics-preserving encoding of specification expressions, and a sound axiomatization of pure methods. In the following, we describe the five dependencies depicted in the figure:

1. The semantics of JML^{--} constructs are only preserved by our encoding if the well-definedness of specification expressions is guaranteed. Therefore, before the encoding of a specification expression, its well-definedness needs to be checked.

2. The well-definedness condition generated for an expression is typically only provable by the use of information contained by sub-expressions. For instance, the well-definedness of "`o != null ==> o.val > 0`" is ensured if `o` is non-null. This is only provable if the premise of the implication can be used. Thus, well-definedness checking relies on the use of the encoding.

3. If a specification expression contains calls to pure methods, as in "`foo() != null ==> foo().val > 0`", then the well-definedness condition of the expression will contain applications of the corresponding uninterpreted function symbols. Such conditions are only provable if the meaning of such symbols is known. Therefore, the axioms over pure methods have to be available when proving the well-definedness of specifications.

4. Axioms that are generated over pure methods should not contain

mathematically unreasonable formulas. Thus, only well-defined expressions should be axiomatized. Therefore, the well-definedness of an expression should be checked before it would get axiomatized.

5. Axioms over pure methods capture the specifications of the methods. Therefore, axioms contain formulas that are encodings of specification expressions. This means that the axiomatization technique relies on the encoding of specification expressions.

Outline. Part I is structured as follows. The next chapter introduces our encoding of specification expressions, as the well-definedness checking and axiomatization techniques rely on it.

In Chapter 5, we show how well-definedness conditions are generated for JML^{--} expressions. At this point, we can already argue that if the conditions are valid then the semantics of JML^{--} expressions is preserved by our encoding.

In Chapter 6, we present our technique for the axiomatization of pure methods. The technique resolves dependencies between specification elements by applying an incremental processing of the elements based on a dependency graph. The proposed technique also includes well-definedness checking, that is, feasibility and well-definedness of specifications is proven at the same time.

Finally, in Chapter 7, we describe how the three techniques were implemented and adapted in the Spec# verification system.

Chapter 4

Encoding of JML^{--} Specification Expressions

In this chapter, we define the encoding of JML^{--} expressions in first-order logic. As discussed in Section 3.1, the main difficulty in defining such an encoding is that the semantics of logical operators has to be preserved, and that state changes made by pure methods have to be handled. This chapter proposes solutions only for the latter problem. The former problem will be resolved in the next chapter, where we present a technique to ensure the well-definedness of specification expressions.

We present two encodings. The first encoding, presented in Section 4.2, takes all possible store changes into account. Although the resulting encoding yields complicated formulas, it can handle JML^{--} expressions without any restriction. The second encoding, presented in Section 4.4, treats pure methods as if they were strongly pure. That is, as if pure methods were not allowed to change the state in any way, not even by the allocation of new objects. This encoding yields significantly simpler formulas than the first one, but it does not come for free: certain admissibility and semantic rules have to be enforced on JML^{--} specifications.

Finally, in Section 4.5, we show that one consequence of the simplified encoding is that the pre- and poststore of a pure method has to be thought of as being identical. This requires further rules to be defined in order to prevent semantic mismatches.

4.1 Encoding of Pure Methods

The Encoding Function. Expressions in programs are always evaluated in terms of a given state, the *current program state*. Since this state is left implicit in the notation, the same expression, for instance `this.f`, may evaluate to different values at different points of the program execution.

The situation is similar with specification expressions in one-tiered spec-

ification languages. Since there is no notion of *current state* in first-order logic, state has to be an explicit parameter of an encoding of JML^{--} expressions. In postconditions, JML^{--} expressions may contain the `\old` and `\fresh` constructs, which allow one to refer to values in the prestate of the specified method. This means that the encoding of postconditions needs two explicit state parameters.

The only part of the state that is relevant for the encoding of specification expressions is the object store. Therefore, we model the state by the **Store** data type introduced in Section 2.2. The signature of our encoding function γ is the following:[1]

$$\gamma : \mathit{Expr} \times \mathbf{Store} \times \mathbf{Store} \to \mathit{Term}$$

The first store argument denotes the "current" store, that is, the store in which the expression is to be evaluated. The second store argument is used only for the encoding of postconditions, and denotes the store of the prestate.

Remark. In the sequel, we will use the convention that if both the "current" store and the prestore needs to be denoted, then the former is denoted by OS' and the latter by OS. If only one store needs to be denoted, then OS will be used. In applications of function γ, we will make it explicit when the second store argument is not used (*e.g.*, in encodings of preconditions) by writing _ at that position, as in $\gamma(\texttt{this.f}, OS, \_)$.

Encoding of Pure Methods. Function symbols that model pure methods take one argument for each parameter of the method, and the object store in which they are evaluated; and yield the result of the method. As a convention, the name of the function symbol that models a pure method in type T will be the name of the method with subscript T, and an additional "hat" symbol ($\widehat{\ }$) on top. In case of overloading, name clashes must be resolved. For constructors, the subscript denoting the enclosing type is dropped.

For example, an instance method m in type T with one implicit parameter (the receiver) and one explicit parameter is modeled by the following function:

$$\widehat{m_T} : \mathbf{Value} \times \mathbf{Value} \times \mathbf{Store} \to \mathbf{Value}$$

In some subtype S of T, the method is modeled by function $\widehat{m_S}$ that (apart from the name) has the same signature as $\widehat{m_T}$.

The modeling of constructors differs only in that the function takes no parameter for the receiver object and the function yields the newly allo-

[1] The encoding of an expression may yield a *Formula*, too. For brevity, we do not distinguish the two cases in the signature.

cated and initialized object. For instance, a constructor C with one explicit parameter is modeled by the function:

$$\widehat{C} : \mathbf{Value} \times \mathbf{Store} \to \mathbf{Value}$$

Remark. Note that this modeling of constructors does not directly follow the semantics of Java, where the **`new`** keyword first allocates a fresh object and then the constructor is called on that object. Our modeling of constructors unifies these two steps. This modeling is justified by the fact that the intermediate state after allocation and before initialization is not observable in specifications given the restrictions and quantifier semantics defined for constructors on page 22.[2]

Remark. Since there is a one-to-one mapping between pure methods and the corresponding uninterpreted function symbols, we will often use them as synonyms in the sequel to abbreviate text. For instance, we say "the specification of a function f" to abbreviate "the specification of the pure method encoded by function f".

Determinism. Encoding pure methods by mathematical functions seems to be justified: according to Assumption 2.2 on page 18, pure methods are deterministic. If this property did not hold then the encoding could possibly lead to unsound reasoning, as theorem provers rely, for instance, on Leibniz's equality: for every values x and y, and function f, if $x = y$ then $f(x) = f(y)$. This would not hold for some pure method that was non-deterministic.

For constructors and methods that return newly-allocated objects, determinism assumes an allocation mechanism that yields objects with the same object identity when called in the same state.

This assumption is valid for Java. Furthermore, the assumption is in line with the underlying programming logic, which models allocation by a function, too—for instance, $OS\,\langle T\rangle$ denotes the store that we obtain after allocating a T-object in store OS.

Object Allocation. In Section 3.1.2, we have seen three examples that demonstrate that state changes made by pure methods may be observable by subsequent specification expressions. These examples show that either such state changes have to be modeled by the encoding or specifications have to be restricted such that state changes become non-observable.

[2] That is, keyword **`this`** may not be mentioned in preconditions and old-expressions, and the range of quantification over allocated objects in the prestate of constructors excludes the object denoted by **`this`**.

4.2 Explicit Modeling of Store Changes

In this section, we will explore the solution that makes the potential store changes of pure methods explicit. For each pure method m of type T, besides function $\widehat{m_T}$, we introduce another uninterpreted function $\widehat{mS_T}$ that yields the store after calling m. The function takes the same arguments as $\widehat{m_T}$. If m has one explicit parameter then $\widehat{mS_T}$ has the signature:

$$\widehat{mS_T} : \textbf{Value} \times \textbf{Value} \times \textbf{Store} \rightarrow \textbf{Store}$$

Analogously, for a constructor C with one explicit parameter, the signature of the function is:

$$\widehat{CS} : \textbf{Value} \times \textbf{Store} \rightarrow \textbf{Store}$$

In the following, we will refer to these functions as *store functions*.

By the use of store functions, expressions can be encoded such that each sub-expression refers to the store resulting from the evaluation of the previous sub-expression. After each method call, the corresponding store function has to be applied and used for the encoding of the succeeding sub-expression.

Remark. In the sequel, we will drop the subscripts in the names of uninterpreted function symbols whenever they are not relevant. For instance, we will use $\widehat{m}$ and $\widehat{mS}$ instead of $\widehat{m_T}$ and $\widehat{mS_T}$, respectively.

Let us revisit the three examples from Section 3.1.2 (pages 33 and 34) to see how the problems get solved by this explicit modeling.

Example 3.2 Revisited. Recall that the example illustrated that if an encoding neglected store changes made by pure methods, then invariants were assumed to hold for objects that were allocated by pure methods. However, this assumption is invalid if allocation was made by a helper method.

Assume `divide`'s precondition, "`v == (new Unsound()).f`", is encoded in store OS. This store is passed to the encoding of the first operand of the `==` operator, as well as for the encoding of the call to constructor `Unsound` in the second operand. This is because the store parameter does not change by the encoding of the parameter name `v`. However, field `f` is not read in OS, but in the store that the store function of method `Unsound` yields: $\widehat{UnsoundS}(OS)$.

The fact that all object invariants hold in prestore OS of `divide` does not imply that the invariant of the new object holds in the modified store. This prevents the invalid assumption, and, thus the soundness problem. □

Example 3.3 Revisited. Recall that the example highlighted the issue of reference comparison being applied on the same operands, which yield fresh objects. If an encoding neglects state changes then static verification and runtime assertion checking yield different results.

Assuming that the encoding begins in store OS, the ensures clause of method `foo` is encoded by the following formula:

$$\widehat{alloc}(this, OS) = \widehat{alloc}(this, \widehat{allocS}(this, OS))$$

This equality is not trivially true, and whether it holds or not depends on the actual behavior of method `alloc`. In our example, the specification of `alloc` implies that the equality does not hold. □

Example 3.4 Revisited. Recall that the issue with the example was that if the pre- and poststore of method `alloc` was considered to be identical, then its ensures clause `\fresh(\result)` would lead to a contradiction.

By denoting the prestore of method `alloc` by OS, the poststore is denoted by store $\widehat{allocS}(this, OS)$. Stating that the return value is not allocated in the former and is allocated in the latter is not a contradiction. Whether the statement holds or not depends on the behavior of the method. □

The Resulting Encoding. To formalize store changes explicitly, the encoding relies on function ω that accumulates store changes made by some expression E. More specifically, ω yields two stores that correspond to the poststore and the prestore after the evaluation of expression E:

$$\omega : Expr \times \mathbf{Store} \times \mathbf{Store} \rightarrow \mathbf{Store} \times \mathbf{Store}$$

The arguments of ω are the same as that of γ. We will write ω_1 and ω_2 to refer to the first (the poststore) and the second (the prestore) components of the result of ω, respectively.

We do not present the definition of encoding function γ for the complete syntax of JML^{--} expressions. Instead, we refer the reader to [35], and here only present two of the more involved cases in order to demonstrate that the encoding leads to complicated and hard-to-read formulas.

Expressions with Binary Operator. The first operand of a binary operator is encoded with the two stores that are passed to γ as arguments. The second operand is encoded in the stores that the encoding of the first operand yields. For instance, the encoding and successor stores of an expression with operator `&&` are the following:

$$\begin{aligned}
\gamma(E \,\&\&\, F, OS', OS) &\triangleq \gamma(E, OS', OS) \;\wedge\; \gamma(F, \omega(E, OS', OS)) \\
\omega(E \,\&\&\, F, OS', OS) &\triangleq \omega(F, \omega(E, OS', OS))
\end{aligned}$$

Method Calls. A call to some pure method m of type T is encoded as a function application of $\widehat{m_T}$. According to the left-to-right evaluation order of Java, the expression that denotes the receiver object is encoded first. Thus, the expression that denotes the first explicit parameter of the method is encoded in the poststore of the encoding of the receiver object. Analogously, the second explicit parameter is encoded in the poststore of the encoding of the first parameter, and so on. Accordingly, the store argument of the application of symbol $\widehat{m_T}$ is the poststore of the encoding of the last parameter.

The successor stores of a method call are expressed using the store function $\widehat{mS_T}$. Its arguments are determined the same way as for function $\widehat{m_T}$. The evaluation of `\old` constructs in the actual parameters of the call to m potentially modifies the method's prestore (for instance, in `obj.m(\old(obj.n()))`). The ω function for method calls changes the second store component of its result accordingly, using the ω_2 function to acquire the second component, the prestore.

For simplicity, we present the encoding and successor stores for a method with one explicit parameter:

$$\gamma(E\texttt{.m(}F\texttt{)}, OS', OS) \triangleq \widehat{m_T}(\ \gamma(E, OS', OS), \gamma(F, \omega(E, OS', OS)), \omega_1(F, \omega(E, OS', OS))\ )$$

$$\omega(E\texttt{.m(}F\texttt{)}, OS', OS) \triangleq (\ \widehat{mS_T}(\ \gamma(E, OS', OS), \gamma(F, \omega(E, OS', OS)), \omega_1(F, \omega(E, OS', OS))\ ),\ \omega_2(F, \omega(E, OS', OS))\ )$$

where T is the static type of expression E.

Example 4.1. Figure 4.1 shows the final and intermediate pre- and poststores that $\omega(\texttt{obj.m(p) \&\& obj.m(\old(obj.n()))}, OS', OS)$ yields. For simplicity, we assume that `obj` and `p` are parameter names. □

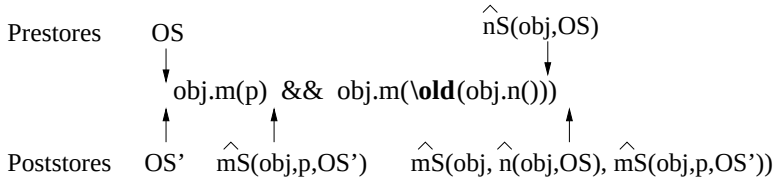

Figure 4.1: Stores yielded by the explicit modeling of store changes

Remark. We do not prove the correctness of the encoding function γ. Such an argument would require, for instance, an equivalence proof against an operational semantics, which would go beyond the objectives of this thesis. Therefore, the encoding function γ can be thought of as our "interpretation" of specification expressions.

4.3 Drawbacks of the Explicit Modeling

The explicit modeling of store changes by the use of store functions eliminated the problems of the encoding for the three examples. However, this modeling has several disadvantages. One direct consequence of applying the store functions is that the encoding clutters up resulting formulas.

Example 4.2. To demonstrate how complex formulas can get, we revisit class `Sequence` in Figure 3.3 on page 36, and partially encode the second ensures clause of method `count`:

$$\gamma(\texttt{!getFirst() == c \&\& rest().isEmpty()}, OS', OS) \equiv$$
$$\neg\widehat{getFirst}(this, OS') = c \ \wedge \ \widehat{isEmpty}(\widehat{rest}(this, G), \widehat{restS}(this, G))$$

where G abbreviates store-function application $\widehat{getFirstS}(this, OS')$.

The resulting formula contains three applications of store functions, two of which form a nested application. The situation gets much worse when the whole postcondition of `count` gets encoded: the method has five ensures clauses and the expressions contained by those clauses get conjoined to form one expression to be encoded. The encoding of that expressions yields a complex and unreadable formula, which makes both automated and interactive verification difficult. □

Besides cluttering up resulting formulas, the use of store functions leads to several other drawbacks.

More Difficult Reasoning Over Stores. Since the encoding yields formulas that contain applications of store functions, we need means to deduce properties of stores that are denoted by store functions.

Since a pure method is assumed not to modify existing objects, we know that every value that is alive in prestore OS of a pure method is alive and unchanged in poststore OS' of the method. We express this property by the predicate $OS \trianglelefteq OS'$, and define it as follows:

$$\begin{aligned} OS \trianglelefteq OS' \ \triangleq \ & \forall X.\ alive(X, OS) \ \Rightarrow \ alive(X, OS') \ \wedge \\ & \forall L.\ alive(obj(L), OS) \ \Rightarrow \ OS(L) = OS'(L) \end{aligned} \tag{4.1}$$

The predicate allows one to relate store functions and their store arguments: for every store function $\widehat{mS}$, store OS, and parameters $p_1, \ldots, p_n$, we know that $OS \trianglelefteq \widehat{mS}(p_1, \ldots, p_n, OS)$ holds.

Although such relations between stores and store functions allow one to reason about objects that existed before calling a pure method, it makes reasoning more difficult. In particular, interactive verification becomes more tedious, and fully automated verification may fail if provers do not find the

proper arguments (among the possibly numerous store parameters) for the predicate.

More Difficult Reasoning for Clients. Store functions make it difficult to match specifications. For instance, clients of method `count` may be interested in the specification expression contained by the third ensures clause, but not in the first or second one. Still, they have to deal with the store changes potentially made by the expressions in the first and second clauses because the corresponding store functions appear in the encoding of the third clause.

Store Changes of Quantifiers. For universal and existential quantification no encoding can be given in the program logic introduced in Section 2.2 that takes all store changes explicitly into account. Consider the following two expressions:

```
(\forall int x. new C(x).b)   and   (\exists int x. new C(x).b)
```

In the first case, there is an infinite number of object allocations[3] and the order in which they take place is unknown. In the second case, we may not know how the store changes as it depends on the actual instantiation.

Losing Commutativity of Operators. Due to the store functions, commutativity of operators is harder to establish because, for instance, expressions `m()==n()` and `n()==m()` are encoded by different formulas.

4.4 Simplified Encoding of Store Changes

To overcome the drawbacks of the explicit modeling of store changes, we propose a simplified encoding of pure methods. The simplified encoding omits the use of store functions, that is, store changes that pure methods possibly make are not taken into account. The main effect of this simplification is that in the encoding of specification expressions, the pre- and poststore of a call to a pure method is not distinguished by the use of store functions, but is considered to be identical. Therefore, application of the ω function (used for accumulating store changes made by sub-expressions) is not needed anymore in the definition of γ.

As a consequence, for every precondition P the resulting formula of $\gamma(P, OS, \_)$ only contains OS as store argument. The same holds for the encoding of invariants. And, for every postcondition Q, formula $\gamma(Q, OS', OS)$ only contains OS' and OS as store arguments.

[3] Or a *large* number of allocations in case Java's semantics of integers is modeled.

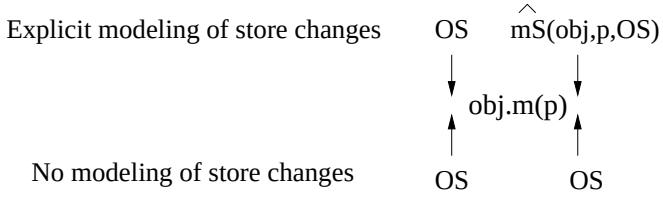

Figure 4.2: Stores with and without the modeling of object allocation

The difference between the explicit and this simplified modeling is illustrated in Figure 4.2, which depicts the pre- and poststores of the encoding of a method call in store OS.

Example 4.3. We revisit Example 4.2 on page 47 and encode the same expression that was shown there:

$$\gamma(\texttt{(!getFirst() == c \&\& rest().isEmpty())}, OS', OS) \equiv$$
$$\neg\widehat{getFirst}(this, OS') = c \ \wedge \ \widehat{isEmpty}(\widehat{rest}(this, OS'), OS')$$

Note that the store arguments in the resulting formula always remain OS'. □

The definition of the simplified encoding is presented in Figure 4.3. Function *FOL*, defined in Figure 4.4, gives the encoding of the operators, formal parameters, and literals of JML^{--} in first-order logic. We assume that the syntax of the underlying logic contains all symbols that appear in the right columns, except of variable names *param*, *this*, $resV$, and the constant symbol *null*.

The simplified encoding eliminates the drawbacks of the explicit encoding. Since the encoding does not use store functions anymore, there is no need to relate stores by the $\trianglelefteq$ predicate, and to use the definition of the predicate to reason about values in poststores of pure methods.

The simplified encoding uses the same prestore and poststore throughout the encoding of an expression, thus the matching of specifications is no longer a problem. For instance, if a client wanted to use the third ensures clause of method `count`, then the encoding of the expression in that clause was not influenced by store changes made by expressions in other clauses. This makes it simpler to match the third clause with other assertions.

As the simplified encoding omits the store functions, commutativity is preserved: for instance, the expressions `m()==n()` and `n()==m()` are now encoded by equivalent formulas.

As demonstrated by the examples in Section 3.1.2, such a simplified encoding, without restrictions, would lead to reasoning that is either unsound

$$
\begin{array}{lcl}
\gamma(E \circledast F, OS', OS) & \triangleq & \gamma(E, OS', OS) \;\; FOL(\circledast) \;\; \gamma(F, OS', OS) \\
\gamma(\texttt{!}E, OS', OS) & \triangleq & \neg\, \gamma(E, OS', OS) \\
\gamma(E\texttt{.f}, OS', OS) & \triangleq & OS'(loc(\gamma(E, OS', OS), f)) \\
\gamma(E\texttt{.m(}F\texttt{)}, OS', OS) & \triangleq & \widehat{m_T}(\; \gamma(E, OS', OS), \gamma(F, OS', OS), OS' \;) \\
 & & \quad \text{where } T \text{ is the static type of } E \\
\gamma(\texttt{new C(}E\texttt{)}, OS', OS) & \triangleq & \widehat{C}(\; \gamma(E, OS', OS),\; OS' \;) \\
\gamma(\texttt{v}, OS', OS) & \triangleq & FOL(\texttt{v}) \\
\gamma(\backslash\texttt{old(}E\texttt{)}, OS', OS) & \triangleq & \gamma(E, OS, OS) \\
\gamma(\backslash\texttt{fresh(}E\texttt{)}, OS', OS) & \triangleq & \\
\multicolumn{3}{l}{\quad \neg\, alive(\gamma(E, OS', OS), OS) \;\wedge\; \gamma(E, OS', OS) \neq null} \\
\gamma((\backslash\texttt{forall}\;\; T\; x.\; E), OS', OS) & \triangleq & \\
\multicolumn{3}{l}{\quad \forall\, x.\; alloc T(x, OS', T) \;\Rightarrow\; \gamma(E, OS', OS)} \\
\gamma((\backslash\texttt{exists}\;\; T\; x.\; E), OS', OS) & \triangleq & \\
\multicolumn{3}{l}{\quad \exists\, x.\; alloc T(x, OS', T) \wedge\; \gamma(E, OS', OS)}
\end{array}
$$

where the shaded parts are added only if the quantified variable is of reference type.

Figure 4.3: Definition of the simplified encoding function γ

or yields results that are not in line with Java's semantics. In order to prevent that, we take a systematic look at the constructs of JML⁻⁻ expressions to find those that are "sensitive" to the omission of store changes. We revise the constructs to see if and how our assumption on pure methods being *deterministic* is effected, and to see if and how our handling of pure methods that *allocate objects* is effected. Once we have an answer to these points, we can define means to make these effects unobservable in specification expressions, thereby, making the explicit and the simplified encoding equivalent.

4.4.1 Revising Determinism

As we have seen in Example 3.3 on page 33, by omitting the possible store changes of pure methods, the encoding of two consecutive calls to a pure method with the same parameters will result in the same term. Unfortunately, this encoding is not faithful to the semantics of Java: values may depend on object identities and object identities are assigned in a state-dependent way by the allocation mechanism. By omitting state changes in our encoding, we cannot model this allocation mechanism faithfully. Thus, our encoding may lead to unexpected evaluations whenever values that depend on the identity of newly-allocated objects are involved.

⊛	*FOL*(⊛)
`&&`	$\wedge$
`||`	$\vee$
`==>`	$\Rightarrow$
`==`	$=$
`!=`	$\neq$
`<`	$<$
`<=`	$\leq$
`+`	$+$
`-`	$-$
`*`	$*$
`/`	*div*
`%`	*mod*

v	*FOL*(v)
`param`	*param*
`\result`	*resV*
`this`	*this*
`null`	*null*
`true`	*true*
`false`	*false*
`-1`	-1
`0`	0
`1`	1

Figure 4.4: Definition of function *FOL*

Therefore, we need to find ways to make the evaluation of specification expressions independent of object identities of newly-allocated objects. To do so, we need to inspect the structure of JML^{--} expressions to determine which constructs make such identities observable. There are three such constructs:

- *reference comparison* (*i.e.*, operators `==` and `!=` with reference-type operands), if both operands refer to a newly-allocated object. This case was demonstrated by Example 3.3.
- *field access*, if the value of the field depends on the identity of a newly-allocated object. As some pure method `foo` is allowed to initialize the fields of newly-allocated objects, expression "`foo().f`" may yield the hash code of a newly-allocated object.
- *method call*, if the return value depends on the identity of a newly-allocated object, such as "`new Object().hashCode()`".

Next, we propose ways to eliminate these cases. We begin with the simpler case of field access and method call.

Proposed Solution for Field Access and Method Call

The problem with field access and method call is caused by the possibility to save or return values that depend on the identity of newly-allocated objects. Therefore, we need a way to ensure that the state of fresh objects is independent of its or other fresh objects' identities after initialization. To achieve this, we forbid to call the `hashCode` method on newly-allocated objects in

implementations of pure methods (including the object being initialized for constructors) and in specification expressions.

One way to check whether this requirement is fulfilled is to apply the equivalence-results methods approach of Leino and Müller [85]. As mentioned earlier, the approach uses self-composition to simulate two executions of the method body, and compares the results against a user-defined equivalence relation. Thus, by defining an equivalence relation that compares return values or performs deep comparison on fields that potentially store object identities, the approach allows one to detect if the use of `hashCode` interferes with the evaluation of specification expressions or the initialization of objects.

If purity checking is realized in a conservative way and calls to non-pure methods are forbidden in implementations of pure methods, then a simple alternative is to consider method `hashCode` to be non-pure. A drawback of this solution is that types such as `Hashtable` cannot be specified since its behavior relies on method `hashCode`, which may not be mentioned in specifications.

Proposed Solution for Reference Comparison

The problem with reference comparison occurs if both operands of operators `==` and `!=` are references to newly-allocated objects. Therefore, a simple way to resolve the problem would be to disallow pure methods to *return* newly-allocated objects—while still permitting the allocation and modification of new objects. Another solution would be to forbid reference comparison in specification expressions. However, both these solutions are too restrictive in many practical cases.

Therefore, we propose a more liberal approach that is based on simple syntactic checks. The checks allow pure methods to return newly-allocated objects as long as they do not lead to the illustrated discrepancy between runtime assertion checking and static verification.

Modifiers. The checks require the use of two modifiers for pure methods, introduced in Section 2.1:

- Modifier **`resultNotNewlyAllocated`** means for a reference-type pure method that the returned object is not newly allocated, that is, it was already allocated in the prestate of the method. Constructors may not be marked with the attribute.

- Modifier **`noReferenceComparison`** means for a pure method or constructor that its specification and implementation does not use operators `==` and `!=` on operands of reference type.

To ensure that the modifiers are applied correctly, the following obligations are posed on pure methods. For methods that are marked with

resultNotNewlyAllocated, the implicit postcondition $alloc(resV, OS)$ is added, where OS corresponds to the prestore of the method. Methods and constructors marked with modifier **noReferenceComparison** (1) may use reference comparison only if one of the operands is the literal **null** and (2) may call methods and constructors in their implementations only if those are marked with modifier **noReferenceComparison**. This can be checked at compile-time.

If the modifier is not present in the signature of a pure method, then it may return fresh objects and may compare references, respectively.

Next, we introduce the notion of an *allocating expression* to describe an expression that may yield a newly-allocated object. Our definition of allocating expressions over-approximates the set of expressions that yield newly-allocated objects; this keeps the analysis sound and simple.

Definition 4.1. ***(Allocating expression)*** *An expression e is considered to be allocating if and only if e is of reference type and is either (a) a constructor call; (b) a method call where the callee is not marked with* **resultNotNewlyAllocated***; or (c) a composite expression with an allocating sub-expression.*

The rationale behind (a) and (b) is self-evident. To see the need for (c), consider a field access `this.m().f` where method `m` is allocating, that is, not marked with **resultNotNewlyAllocated**. Then `m` might return a newly-allocated object whose field `f` refers to a newly-allocated object, too.

Checks on Specifications. The restrictions our checks enforce on specification expressions are the following: (1) operators `==` and `!=` may have at most one operand that is allocating; (2) if two or more parameters (possibly the receiver) of a method call are allocating, then the callee must be one that is marked with **noReferenceComparison**. The latter requirement rules out indirect reference comparisons "hidden" by the implementations of pure methods.

To ensure soundness of the checks in the presence of subtyping, we require an overriding method to be at least as restricted in what its implementation is allowed to do as the overridden counterpart. That is, if an overridden method is marked with one of the attributes then the overriding method must be marked with that attribute, too.

Limitations. There are two main limitations of our design: methods need to be annotated manually by users, and the syntactic nature of our analysis leads to over-approximation. A more accurate analysis that goes beyond syntactic checks, for instance, a points-to analysis could lessen these limitations. Most annotations could be inferred and more methods could be annotated with **noReferenceComparison**.

4.4.2 Revising Object Allocation

After having analyzed and resolved the issues with determinism, we now look at the possible issues with object allocation. As seen in Section 2.2, the store can only be observed by two operations: the lookup function $\_(\_)$ and function *alive*. Thus, we go through the structure of JML⁻⁻ expressions to find all constructs whose encoding uses any of these two functions. Such constructs possibly invalidate the desired equivalence of the encoding that takes all store changes into account and the simplified encoding. In such cases, certain restrictions have to be made in order to establish the equivalence of the two encodings.

Remark. The analysis implicitly assumes that allocation made by a sub-expression of primitive type does not have an effect on the evaluation of a succeeding sub-expression E, provided that E does not contain quantification.[4] For instance, in expression `new C().value == foo()+10`, the allocation made by the constructor is not observable by the expression on the right-hand side of the equality.

Intuitively, the validity of the assumption follows (1) from the syntax of JML⁻⁻ expressions, which does not contain constructs (*e.g.*, the let-expression) to store references to newly-allocated objects for later use; and (2) from the restriction we make on the use of hash code, proposed above. We do not argue for the validity of the assumption: a formal proof would require the introduction of additional formalizations that are not needed in the rest of the thesis, while an informal argument, based on the two points above, is straightforward.

Base case: In the base case we have parameter identifiers and literals. Since their encoding does not contain an application of the lookup function or function *alive*, the omission of object allocation is not observable.

Step case: Next we look at compound expressions and assume that store changes are not observable in sub-expressions.

• **Binary operators, negation, and old-expression**: the encoding of these constructs does not contain functions that observe the store, thus by the assumption on sub-expressions we can conclude that omission of object allocation is not observable for these constructs.

• **Method and constructor call**: a call to a pure method or constructor is encoded by an application of an uninterpreted function symbol. Although the function application neither explicitly uses the lookup nor the *alive*

[4] If E contains quantification, then the objects allocated by the preceding sub-expression are in the scope of the quantifier and thereby may have an effect on the evaluation of E.

function, the actual meaning of the function application is given by the axiom that is generated over the symbol.

The precise form of generated axioms is presented in Chapter 6, here we just give the intuition behind the axioms. An axiom over a pure method essentially expresses the semantics of invariants and pre- and postconditions, as defined in Section 2.1. Moreover, an axiom contains properties that the semantics of the underlying programming language guarantees, such as the allocatedness of the receiver object and the aliveness of the return value.[5] The general form of an axiom generated over some pure method m with one explicit parameter is the following:[6]

for every prestore OS_{pre} and poststore OS_{post}, and
for every receiver object o allocated in OS_{pre} and parameter alive in OS_{pre},
  if all invariants for all objects allocated in OS_{pre} hold and
    the precondition holds for o in OS_{pre},
  then the return value is alive in OS_{post}, and
      all invariants for all objects allocated in OS_{post} hold, and
      the postcondition holds in OS_{post}

(4.2)

The lookup function is not directly used in (4.2). Still, the function may be applied in the encodings of invariants or pre- and postconditions. However, by the time axioms are generated, problematic specification expressions are eliminated by the restrictions that our analysis imposes on specification expressions. Therefore, we can say that store changes are not observable due to the use of the lookup function.

In the formalized version of (4.2), the *alive* function is used wherever allocatedness and aliveness is mentioned above. Therefore, this is a case we need to consider.

As explained above, a precondition is entirely encoded using a single store. Regarding (4.2), this means that even if the precondition of m allocated an object, the store would remain OS_{pre}, thereby extending the scope of invariants over the fresh object. The same applies for postconditions and invariants, too.

This "unnoticed" extension of the scope of invariants may cause trouble if the fresh object violated the invariant. An example for this situation was Example 3.2 on page 33 where the scope of invariants was extended over the (violating) newly allocated object.

This can only happen if the violating allocation was made by a pure helper method, which are the only methods that are not required to establish the invariants.

[5] Recall from Section 2.2, only objects may be *allocated*, while any value may be *alive*.

[6] For constructors, the object being initialized is excluded from the scope of invariants in OS_{pre}. For helper methods, all objects are excluded both in OS_{pre} and in OS_{post}.

Proposed Solution. To prevent the issue with helper methods, we forbid the use of helper constructors and helper methods that are not marked as `resultNotNewlyAllocated` in the specification of non-helper methods. Simple static checks can enforce this requirement.

Remark. Methods that are not marked as helper but violate invariants (according to their postconditions) are caught by the axiomatization technique that we propose in Chapter 6: One of the requirements of the technique is that for each pure method a witness has to be exhibited for the satisfiability of the postcondition under the assumption that the invariants hold.

Remark. Another consequence of considering the pre- and poststore of a call to some pure method identical is that in (4.2), stores OS_{pre} and OS_{post} are considered to be identical when the axiom is used for reasoning about the call. This means that if the simplified encoding is applied, then special care has to be taken when writing specifications for pure methods. We postpone the discussion of this situation until Section 4.5, where the consequences are analyzed in detail.

• **Fresh-expression**: a fresh-expression specifies in postconditions that a given object was not allocated in the *prestore*. Therefore, whether allocation in the poststore is modeled or not does not have an effect on the evaluation of fresh-expressions.

• **Field access**: a field access is encoded by an application of the lookup function. When store changes are explicitly modeled, the encoding looks as follows:

$$\gamma(E.\mathtt{f}, OS', OS) \triangleq \omega_1(E, OS', OS)(\gamma(E, OS', OS).\mathtt{f})$$

where $\omega_1(E, OS', OS)$ denotes the poststore of the encoding of E.

In the simplified encoding store changes are not modeled, thus the poststore of E is always OS'. Intuitively this would be wrong if E denoted a newly allocated object, because the object (*i.e.*, the value denoted by $\gamma(E, OS', OS)$) would not be alive in OS'. In such cases, for instance, a field access on E would not be meaningful.

However, by the fifth line of (4.2), one can deduce that the return value of a pure method is alive in the poststore. Therefore, if expression E is a call to a method that yields a newly allocated object, we can deduce by (4.2) that the object is alive in OS'. Therefore, a field access on E is meaningful.

Although this way of reasoning with (4.2) might seem counter-intuitive, it can be thought of as the modeling of an allocation mechanism in which (1) the store contains all objects that the program at hand is ever going to allocate, (2) unallocated objects are already in an initialized state, and (3)

the role of a constructor call is merely to select one of these pre-fabricated objects and make it "alive".

Using this kind of allocation mechanism for our analysis is justified by the following argument. There is no way specification expressions can observe the intermediate state of the (usual) object allocation mechanism in which the new object is created but not yet initialized. Thus, the only observable difference between the two allocation models is that in our model yet non-allocated objects are present in the store and are already initialized. However, in both models such objects are non-alive. And this is the key point, as the evaluation of formulas that result from the encoding only depends on objects that are alive. As a consequence, whether a non-alive object is initialized or not does not influence the evaluation of formulas.

Remark. A consequence of the allocation mechanism discussed above is that one of the axioms that Poetzsch-Heffter and Müller [111] define for the store operations introduced in Section 2.2 is invalidated. The axiom states that each field of a non-allocated object has the zero-equivalent value of its type (*i.e.*, `null`, 0, or `false`). Therefore, this axiom should not be used together with the encoding of pure methods that we propose.

• **Quantification**: according to the semantics given in Section 2.1, quantification over variables of reference type ranges over allocated objects. In case the quantified expression allocates new objects for some instantiation, conceptually, the set of allocated objects gets larger. On the other hand, the simplified encoding omits store changes made by specification expressions, therefore, the set of allocated objects remains the same throughout the evaluation of a given quantification. We resolve this conflict by precisely specifying the store in which quantification is evaluated.
Proposed Solution. As discussed in Section 4.3, store changes made by quantified expressions cannot be explicitly modeled; not even if we wanted to. Therefore, we define the semantics of quantification such that every instantiation of the quantified expression is evaluated in the same state, namely, in the state in which the evaluation of the quantification began. That is, the interpretation of quantification does not take into account objects that are allocated by the quantified expression.

4.5 Encoding of Pure-Method Specifications

Thanks to the simplified encoding, for every expression and store arguments, $\gamma(E, OS', OS)$ yields a formula that only contains OS and OS' as heap arguments. We have seen earlier that this has several advantages.

As mentioned above, a consequence of the simplified encoding is that the pre- and poststores of pure methods are considered to be identical when

reasoning about calls to them in specifications. In this section, we show that this "merging" of pre- and poststores has to be taken into account when writing specifications for pure methods. The reason is that not only potential allocation made by specification expressions is omitted, but also potential allocation made by the *implementation* of the method at hand. This is depicted in Figure 4.5 in terms of a Hoare-triple with precondition P and postcondition Q.

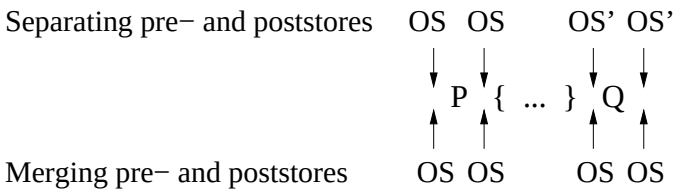

Figure 4.5: Merging the pre- and poststore of a pure method

In this section, we analyze the constructs of JML⁻⁻ expressions to see if there are undesired effects of "merging" the pre- and poststores. And, if there are any, we propose solutions to eliminate the undesired effects.

Remark. The pre- and poststore of a pure method is only considered to be identical when reasoning about calls to the method *in specification expressions*. When reasoning about the *source code* of the pure method, the pre- and poststore is not considered to be identical. For instance, consider a pure method that allocates and initializes a new object o. In order to prove the implicit "**`assignable \nothing`**" clause of the method that permits only the mutation of newly allocated objects, one has to prove that o is indeed fresh. This would not be possible if store changes made by the method were not taken into account.

Remark. It is important to note that considering the pre- and poststores of a pure method to be the same has great advantages, too. Axioms generated over pure methods only need to quantify over one store variable; instead of two, as indicated by (4.2). As a consequence, axioms do not need to relate pre- and poststores by the $\trianglelefteq$ predicate.[7] Clearly, such axioms lead to simpler reasoning over pure methods.

Revising Determinism. As seen in Section 4.4.1, problems only occur if the identity of two fresh objects are compared and the two expressions that denote these objects are encoded by the same term in the underlying logic. An example was expression `alloc()==alloc()` in Figure 3.2 on page 34.

If pre- and poststores are considered to be identical, then the main difference interesting for this case is that the **`\old`** construct does not have any

[7] For simplicity, this relation is not included in (4.2). Our previous work shows how the relation can be formally integrated into axioms [35].

effect. Consequently, the encoding of expression `\old(alloc())==alloc()` would also yield an undesired formula. However, the checks proposed in Section 4.4.1 eliminate such specifications, too. Therefore, regarding determinism, we need not take further measures.

Revising Object Allocation. As seen in Section 4.4.2, the only cases where allocation may lead to problems are method calls, fresh-expressions, and quantification. Therefore, we only need to consider these cases.

• **Fresh-expression**: if pre- and poststores are considered to be identical, say OS, then for every expression E the encoding of the `\fresh` construct (see Figure 4.3 on page 50) yields the following:

$$\neg\, alive(\gamma(E, OS, OS), OS) \;\wedge\; \gamma(E, OS, OS) \neq null$$

This formula always evaluates to **false**: the object denoted by E should be non-alive and non-null at the same time in store OS, however, there is no reference-type JML^{--} expression that could denote such an object.

This is not faithful to the semantics of the construct as pure methods may allocate new objects that can be referred to in fresh-expressions. In such cases the construct is expected to yield **true**. Example 3.4 on page 34 demonstrated such a situation.

Proposed Solution. The only way to resolve this semantical mismatch is to forbid the use of fresh-expressions in the specifications of pure methods.

This restriction rules out, for instance, postcondition `\fresh(\result)` of method `alloc` in Figure 3.2 on page 34. This leads to incompleteness is certain cases, for example, expression `\fresh(alloc())` cannot be proven.

In the specification of pure methods, the only useful argument of `\fresh` is `this` in postconditions of constructors, `\result` in postconditions of methods, and expressions that denote fresh objects reachable through these keywords, such as `\result.f`. These are the only expressions that may denote objects that are both newly-allocated and observable in specifications. Therefore, the only consequence of forbidding `\fresh` in pure-method specifications is that clients cannot deduce whether objects denoted by such expressions are fresh or not, and thereby different from every other objects. Such knowledge may be useful, for instance, to reason about the effects of a field update on the object returned by a pure method.

• **Quantification**: The scope of quantifiers with bound variables of reference type spans over allocated objects. Therefore, if a pure method allocates new objects in its implementation, the scope of a quantifier gets larger in the postcondition. However, by considering pre- and poststores identical, these fresh objects "disappear" from the scope of quantifiers in postconditions. To illustrate that this may be observable, consider the class in Figure 4.6.

```
class C {
  int val;

  pure C singlePositive()
    requires (\forall C c. c.val == 0);
    ensures (\exists C c. c.val > 0);
  {
    C temp = new C();
    temp.val = 5;
    return temp;
  }
}
```

Figure 4.6: Limited quantifier expressivity in pure-method specifications

Clearly, the postcondition of method `singlePositive` holds because the method returns a fresh object that satisfies the existential quantifier. However, if the pre- and poststore of `singlePositive` is considered to be the same, then from the precondition one would expect the postcondition to always evaluate to **false**.

Proposed Solution. To "resolve" this issue, we define the semantics of quantifiers for pure methods so that the store in which they are evaluated is always the prestore of the method being specified. As a consequence, newly-allocated objects are not taken into account by the evaluation of quantifiers in postconditions. Obviously, this semantics does not resolve the problem but at least eliminates the semantical mismatch by precisely defining the scope of quantifiers.

As the example above illustrates, this quantifier semantics limits expressivity: the intuitively correct specification and implementation of method `singlePositive` is deemed incorrect.

Remark. Generating an axiom from a contradicting pair of pre- and postcondition (such as that of method `singlePositive` in the proposed quantifier semantics) would lead to inconsistency. The approach presented in Chapter 6 rules out such cases, thus, there is no need to define any restriction here.

• **Method Call**: As mentioned in Section 4.4.2, we need to consider the consequences of stores OS_{pre} and OS_{post} being identical in (4.2). To do so, we have to consider the three properties that are stated over poststore OS_{post} and analyze their meaning in prestore OS_{pre}:

1. The return value is alive in the prestore, OS_{pre}. Clearly, the statement is not true if the return value is a reference to a newly-allocated object. However, the statement could only lead to a contradiction if there was a way to specify that the return value was not allocated in the prestore. By having

forbidden the use of **fresh** in pure-method specifications, this property cannot be specified with the syntax of JML^{--}.

2. All invariants hold in prestore, OS_{pre}. As invariants quantify over all allocated objects, it seems that by using store OS_{pre} we lose the information that invariants hold for objects that are allocated by specification expressions and by the method at hand. However, for the former set of objects the loss of information is not a problem in practice as these objects are not used when reasoning about source code—there is even no way to refer to them. For the latter set of objects the information can be restored as follows. By 1., we know that the return value is *alive* in OS_{pre}. Thus, if we can additionally derive (for instance, from the postcondition) that the value is of reference type and non-null, then we can deduce that the object is *allocated* in OS_{pre}. Therefore, the scope of invariants spans over that object. Aliveness of values reachable from the alive result value can be deduced by one of the axioms of Poetzsch-Heffter and Müller's program logic [111]. Allocatedness of reachable objects can be deduced the same way as for the return value described above.

3. The postcondition holds in the prestore, OS_{pre}. We already went through the structure of JML^{--} constructs to see and fix the undesired consequences of merging pre- and poststores. Therefore, we can assume that no sub-expression of postconditions can cause trouble.

Since none of the three cases lead to discrepancy with the expected semantics or to contradiction, there are no measures to take for method calls.

This concludes the analysis for object allocation.

4.6 Summary

To sum up the results of this chapter, a quick overview of the consequences of the proposed simplified encoding is given. We list the necessary restrictions, how they can be enforced, and alternatives to the restrictions, if any.

• Method `hashCode` may not be called on newly-allocated objects in the specification and implementation of pure methods.
Enforce: either by proof obligations (*e.g.*, approach of equivalent-results methods [85]) or by considering `hashCode` to be non-pure and forbidding non-pure calls in the implementation of pure methods. The latter solution is an over-approximation and rules out many "harmless" implementations, because non-pure methods may be called in implementations of pure methods as long as they only alter the state of newly allocated objects.

• References to fresh objects may not be compared in specifications.
Enforce: by restricting operators `==` and `!=` to have at most one allocating

operand, and requiring every pure method that is ever called with two or more allocating parameters to be marked with **noReferenceComparison**. *Alternative*: a crude over-approximation is to forbid pure methods to return fresh objects (while allowing allocation).

• Specifications of non-helper methods may not call helper constructors and helper methods that are not marked as **resultNotNewlyAllocated**. *Enforce*: by simple syntactic checks.

• The semantics of quantifiers needs to be defined such that the store in which they are evaluated is the prestore in preconditions and inside old-expressions, and the poststore otherwise. That is, store changes possibly made by preceding sub-expressions and the quantified expression are not taken into account.

Consequences for Pure-Method Specifications

• Freshness of objects allocated by pure methods may not be specified. *Enforce*: by forbidding fresh-expressions in specifications of pure methods.

• The semantics of quantifiers needs to be defined such that they are always evaluated in the prestore, even if the quantification occurs in a postcondition.

Note that different semantics was proposed for quantification depending on whether it appears in the specification of a mutating method or a pure method. This might be confusing and undesired. However, for the reason explained above, the semantics for pure methods cannot be relaxed if the simplified encoding is used. Therefore, there are only two ways to make the quantifier semantics uniform for mutating and pure methods.

1. One can forego using the simplified encoding. This makes the quantifier semantics more expressive for pure methods (*cf.* the specification of method `singlePositive` in Figure 4.6). On the other hand, formulas that correspond to encoded specification expressions as well as reasoning about pure methods gets more complex.

2. One can define the quantifier semantics for mutating methods the same way as it is defined above for pure methods. While this makes specifications of mutating methods less expressive, reasoning over pure methods becomes significantly simpler due to the simplified encoding.

Alternatively, new syntax and semantics could be introduced for quantifiers as follows:[8]

3. One could introduce two different kinds of quantifiers to represent the two kinds of semantics described above.

[8]These alternatives were proposed by Gary T. Leavens in private discussions.

4. One could introduce quantifiers with an explicit parameter that specifies the store in which the evaluation of the quantifier is meant. Due to the simplified encoding of pure methods, the parameter would be restricted to be the prestore in specifications of pure methods.

4.7 Related Work

Encodings of one-tiered specification languages commonly model pure methods as uninterpreted function symbols [32, 91, 65, 122]. However, according to our knowledge, our work is the first encoding that explicitly addresses the issues of weakly-pure methods. In the following, we summarize the way encoding of pure methods is done in program-verification techniques and tools that use one-tiered specification languages. We briefly introduce each of these techniques and tools as they will be referred to in latter chapters, too.

Program Verifiers. ESC/Java [47] was a tool developed for the cost-effective extended static checking of Java programs. ESC/Java used a specification language similar to JML, and was designed to catch bugs within a few seconds. The tool was designed to be unsound in certain well-defined domains, for instance, in the handling of loops. ESC/Java did not allow method calls to occur in specification expressions, thus the issues discussed in this chapter were trivially eliminated.

Its successor, ESC/Java2 [70] supports JML as specification language, thus allows the use of pure methods in specification expressions. The tool implements Cok's formalization [32] for their handling. However, the formalization does not handle the aspect of object allocation, therefore, the unsoundness and discrepancy illustrated by the three examples in Section 3.1.2 are not prevented by his approach, and thus, ESC/Java2.

Why [43, 46] is a verification platform that provides an intermediate programming and specification language, and a verification condition generator that interfaces several theorems provers, both automatic and interactive (*e.g.*, Simplify, Z3 and PVS, Isabelle). Krakatoa [91] and Caduceus [45] are tools that utilize the Why platform for the verification of annotated Java and C programs, respectively.

The specification languages used by Krakatoa and Caduceus are derived from JML, but some constructs are removed while others are added. The main restriction interesting for us is that methods may not be called in specifications. For the purpose of abstraction, users can introduce new logical sorts, function and predicate symbols, and axioms to define the meaning of the symbols [90]. The axiomatization can be directly done in the theorem prover used for formal reasoning. This may be useful in cases when something is not expressible on the JML level or is more convenient to express in the theorem prover.

Applications of such logical function and predicate symbols may occur in specifications, and since they are purely logical, the issues with object allocation are prevented.

The KeY System [15] is a formal approach to the development of Java and Java Card programs. It aims to integrate the phases of design, implementation, formal specification, and formal verification. In the design phase, users can make use of UML diagrams formally specified by the Object Constraint Language (OCL) [109], a standardized specification language specifically tailored to UML. Actual implementations can be verified against the UML/OCL specification as well as against specification written in JML. The underlying logic of the KeY System is first-order Dynamic Logic [58, 59], and the tool provides its own theorem prover based on taclets [14].

The KeY System considers pure methods to be strongly-pure (both in OCL and JML specifications), which is enforced by proof obligations. Although this makes sure that the problems discussed in this chapter are eliminated in KeY, it also makes the expressivity of pure methods limited [103, 118]. This is true for encodings of OCL specifications in general (*e.g.*, that of HOL–OCL [24]) as purity in OCL means strong purity.

Jack [26] is a program verifier for JML annotated Java and Java Card programs. Jack uses a weakest precondition calculus to generate proof obligations. The proof obligations are represented in an intermediate language, in the Java Proof Obligation Language (JPOL). JPOL is a mixture of Java and JML constructs, constructs of set theory, and a number of "ad-hoc" constructs. From the JPOL representation Jack can generate input for a number of fully automatic and interactive theorem provers in order to discharge proof obligations.

Although Jack does not explicitly model store changes made by pure methods, postcondition `alloc() == alloc()` in Example 3.3 on page 33 is correctly not provable by Jack. The reason is that the encoding of the two calls yields two different logical variables, therefore the postcondition cannot be trivially proven as the terms on the two sides of the equality operator are different. Reasoning about the equality requires the use of the axioms generated for method `alloc`. Therefore, this approach of introducing separate logical variables for the calls resolves the issue of reference equality over fresh objects.

Jack does not support the **`helper`** modifier and the **`\fresh`** construct, thus the issues of the other two examples presented in Section 3.1.2 cannot occur in Jack.

The Spec# system (described in detail in Chapter 7) supports weakly-pure methods in contracts. Spec# applies the simplifications proposed in this chapter for the encoding of specification expressions and for the axiomatization of pure methods.

The issue with reference comparison is resolved by the checks described in Section 4.4.1 [33]. Spec# considers method `GetHashCode` to be pure, and

does not ensure that the method is not called on newly-allocated objects. Therefore, it is possible to prove that the hash code of two newly-allocated objects are equal. However, this is due to the fact that the technique of equivalent-results methods is not yet implemented in the tool.

Spec# does not have the notion of helper methods and it is not possible to initialize a fresh object such that its invariant is broken at the end of the initialization. This is because the invariant of the initialized object is always checked at the end of the constructor (even if it is marked as `[NoDefaultContract]`). Therefore the problem of assuming a postcondition that contradicts invariants cannot occur.

In contrast to our proposal, specifications of pure methods in Spec# may specify the freshness of objects. However, consistency checks ensure that no axiom is extracted from such specifications (recall from Section 4.5 that such axioms would lead to semantical mismatches). Finally, the issues with quantification over objects is prevented by only allowing quantification over the elements of some collection. Since a pure method may not modify a collection that existed in the prestore, the scope of quantifiers is always the same in the pre- and poststore of a pure method.

Jacobs and Piessens [65] introduce *inspector* methods. Inspector methods are essentially pure methods with defined read effects: an inspector method may read the state of the receiver object and parameters marked with **`state`**, and the state of objects that are owned by the receiver object or **`state`**-parameters. Inspector methods may not have postconditions and their method bodies must be of the form `{ return` E`; }`, where E is a side-effect free expressions, which may not contain object and array creation. Thereby, the issues with allocation is prevented.

Dynamic Frames. In order to reason about programs, a method specification has to define the *frame property* of the method: which part of the state is potentially modified, and which is left unchanged by the method [21]. The specification of frame properties is difficult in programming languages that support modular development and information hiding because the set of all program variables are not known at the time of developing a module, and specifications may not reveal variables used by implementations. Therefore, different techniques were developed to express frame properties in an abstract way, for instance in [80, 86, 100, 114].

A recent approach is the use of *dynamic frames* proposed by Kassios [69]. Dynamic frames provide a more flexible solution than previous approaches in that they come with no restrictions on object structures such as alias control [101]. In the following we list approaches that use pure methods to express dynamic frames.

Smans *et al.* [128] developed a verification technique that extends method specifications with explicit **`reads`** clauses for pure methods and **`writes`** clauses for mutating methods. The clauses are used to specify read and write

effects, and are given in terms of special pure methods, *dynamic frames*, that return sets of locations. In [127], Smans *et al.* improve the technique by inferring read and write effects. The approach of dynamic frames is applied in VeriCool [126], a verification technique and tool developed for the handling of concurrent programs. Common in all three approaches, and relevant for the discussion, is that expressions that get encoded to the underlying logic are strongly-pure.

Dafny [81] is an object-based language with specifications that are based on dynamic frames. Dafny does not allow specifications to contain calls to methods, but supports abstraction via specification-only functions. The meaning of such functions is given by strongly-pure expression, thus the problems of object allocation in specifications are eliminated.

Ballet [122] is a verification tool for Eiffel programs. It translates Eiffel programs and their specifications to the intermediate language of Spec#. Similarly to the work of Smans *et al.* [128], Ballet expresses dynamic frames in terms of pure methods. Ballet allows specifications to contain pure methods but the encoding does not take object allocation into account, therefore, the tool does not resolve the problematic cases presented in this chapter.

Model Fields. Model fields are similar to parameterless pure methods in that their values are not looked up in the store, but depend on the state of objects. In JML, the relation between the value of a model field and the current state is specified by a **`represents`** clause. In general, such relations may contain calls to pure methods, thus the encoding of model fields raises similar issues to the ones the encoding of pure methods raises.

Breunesse and Poll [22] describe a way to encode model fields as pure methods. The encoding replaces every model field by a pure method whose postcondition corresponds to the **`represents`** clause of the model field. The issues with weakly-pure expressions in **`represents`** clauses are not addressed.

Work by Leino and Nelson [79, 86], and Müller [100] provides a sound and modular handling of model fields using different techniques. However, neither of the two approaches allow representation functions for model fields to include method calls, thus preventing the need for handling side-effects.

Work by Leino and Müller [84] extends the Boogie methodology [83] to handle model fields in a sound, modular, and practical way. One of the main novelties of the approach is that the values of model fields are computed and saved in the store, just like the values of ordinary fields. Therefore, reading a model field does not have side-effects because the (possibly weakly-pure) representation relation is not applied. For instance, for a model field m, $m == m$ yields true, even if the representation for m allocates a new object. Leino and Müller's approach handles weak purity for model fields, but not for method and constructor calls, in general.

Chapter 5

Well-definedness of Specification Expressions

Programming errors often lead to runtime exceptions. Typical errors include division by zero or dereferencing `null`. In specification languages, the same mistakes may occur in specification expressions, too. When such bogus specification expressions are executed during runtime assertion checking, then exceptions are thrown just as if the expressions would appear in program code. However, during static verification, specifications are not executed but encoded in logical formulas. Thus, the question arises how an encoding should handle such bogus expressions—the encoding function γ did not address this issue at all.

In this chapter, we answer this question by discussing the different alternatives and by giving our preferred solution for the expression syntax of JML^{--}. We propose an operator that generates verification conditions for JML^{--} expressions. The validity of such conditions guarantees that the corresponding expressions are free of the kind of mistakes mentioned above.

At the end of the chapter, we explain the difficulty of proving such conditions in a setting like ours, where the presence of pure-method calls in specification expressions make specification elements dependent on each other. The technique we propose to overcome this difficulty is then presented in Chapter 6. That is, the purpose of this chapter is to introduce the operator that is used in the sequel for the well-definedness checking of specifications.

5.1 Background

Formally, the problem with the bogus expressions mentioned above is that they contain *partial operations* that are, by mistake, applied outside their domains. In such cases the expressions are said to be *ill-defined*. The treatment of ill-defined expressions is a well-studied topic in the literature as most formal approaches allow the use of partial operations in specifications.

5.1.1 Various Approaches to the Handling of Ill-definedness

We briefly summarize three mainstream approaches to the handling of ill-definedness. For detailed accounts, the reader is referred to the work of Abrial and Mussat [4, Section 1.7], which discusses further approaches; to Arthan's paper [6], which classifies the different approaches to undefinedness; and to Hähnle's survey [55] on the use of many-valued logic for the modeling of partiality in different specification languages.

Three-valued Logics. One of the standard approaches to handle ill-defined expressions is to define a three-valued semantics [71] by considering ill-defined expressions to have a special value, *undefined*, denoted by $\perp$. For instance, expressions `null.f` and `x/0` are both considered to evaluate to $\perp$. In order to reason about specifications with a three-valued semantics, undefinedness is lifted to formulas by extending their denoted truth values to $\{\mathbf{true}, \mathbf{false}, \perp\}$. This leads to the development of three-valued logics, which thus fully integrate ill-defined expressions into the formal logic. This approach is attributed to Kleene [71]. A well-known three-valued logic is LPF [13, 30] developed by Jones *et al.* in the context of VDM [68]. Other specification languages that follow this approach include Z [129] and OCL [109].

A well-known drawback of three-valued logics is that they may seem unnatural to proof engineers. For instance, in LPF, the law of the excluded middle and the deduction rule (a.k.a. ImpI) do not hold. Furthermore, a second notion of equality (called "weak equality") is required to avoid proving, for instance, that formula $x/0 = y/0$ holds. Another major drawback is that there is significantly less tool support for three-valued logics than there is for two-valued logics.

Underspecification. The approach of underspecification assigns to an ill-defined expression a definite, but unknown value from the type of the expression [50]. Thus, the resulting interpretation is two-valued, however, in certain cases the truth value of formulas cannot be determined due to the unknown values. For instance, the truth value of $x/0 = y/0$ is known to be either **true** or **false**, but there is no way to deduce which of the two, since the actual values of $x/0$ and $y/0$ are unknown. However, for instance, $x/0 = x/0$ is trivially provable. This might not be a desired behavior. For instance, the survey by Chalin [27] argues that this is against the intuition of programmers, who would rather expect some kind of error to occur in the above cases, which would also be the behavior of a runtime assertion checker.

Underspecification is applied, among others, in the Isabelle theorem prover [106], the Larch family of specification languages, and in the KeY System. The semantics of JML used underspecification until 2007.

Eliminating Ill-definedness. A common technique to reason about specifications with a three-valued semantics is to *eliminate* ill-defined expressions before starting the actual proof [16, 78, 4, 17]. *Well-definedness conditions* are generated, whose validity ensures that all formulas at hand can be evaluated to either **true** or **false**. That is, once all well-definedness conditions have been discharged, $\perp$ is guaranteed to never be encountered.

The advantage of the technique is that both the well-definedness conditions and the actual proof obligations are to be proven in classical two-valued logic, which is simpler, better understood, more widely used, and has better automated tool support [130] than three-valued logics.

The technique of eliminating ill-defined expressions in specifications by generating well-definedness conditions is used in several approaches, for instance, in Event-B, PVS [110], CVC Lite [12], and ESC/Java2. Since 2007 the semantics of JML also requires specification expressions to be well-defined [77].

Our Choice of Technique

The approach we follow in this thesis to cope with the problem of partiality is the technique of eliminating ill-defined expressions. That is, before further analysis or processing of some specification element, we first generate a well-definedness condition for it, and if the condition is not provable, then the specification element gets rejected.

Our preference for this technique is due to the main drawbacks of the other two techniques:
(1) General-purpose theorem provers commonly used for the automatic and interactive verification of programs typically use two-valued logics. For instance, Simplify [38], Z3 [36], Isabelle [106], or Coq [20]. These provers could not be applied in case one decided for the use of three-valued logics.
(2) As pointed out by Chalin [27], the assertion semantics that the technique of underspecification yields goes against the expectations of programmers. Furthermore, it also leads to discrepancy between runtime assertion checking and static verification.

In the following section, we summarize the most relevant work on the technique of eliminating ill-definedness.

5.1.2 The Approach of Eliminating Ill-definedness

The first approach for the elimination of ill-definedness is attributed to Hoogewijs, who introduced the Δ operator in the form of a logical connective in [63], and proposed a first-order calculus, which included this connective. Later, Δ was reformulated as a formula transformer $\mathcal{D}$, for instance, in [16, 4, 17]. $\mathcal{D}$ takes a (possibly open) formula φ and *domain restriction*

δ, and produces another formula $\mathcal{D}(\varphi)$.[1] The interpretation of $\mathcal{D}(\varphi)$ in two-valued logic is **true** if and only if the interpretation of φ in three-valued logic is different from $\bot$.

Domain Restrictions. Domain restriction δ is a mapping from a set of function symbols $\mathbf{F}$ to formulas. For each function $\mathbf{f} \in \mathbf{F}$, formula $\delta(\mathbf{f})$ characterizes the domain of function $\mathbf{f}$ and may contain k free variables where k is the arity of $\mathbf{f}$. For instance, for the division operator, the domain restriction δ requires the divisor to be non-zero. In text, we will refer to such formulas as the domain restrictions of the corresponding symbols, for instance, the "domain restriction of the division operator".

Besides properly characterizing the domains of functions, another important requirement on domain restriction δ is that for each $\mathbf{f} \in \mathbf{F}$ formula $\delta(\mathbf{f})$ is well-defined. That is, $\delta(\mathbf{f})$ does not evaluate to $\bot$ itself.

These two main properties of domain restrictions can be defined as follows in terms of structures and the semantical operation **wd** [17].[2]

Definition 5.1. *Domain restriction δ characterizes structure $\mathbf{M}$ and is well-defined for variable assignment θ, denoted by $(\mathbf{M}, \theta)^\delta$, if and only if:*

1. *δ characterizes the domains of function interpretations for $\mathbf{M}$. That is, for each $\mathbf{f} \in \mathbf{F}$ and $\mathbf{l}_1, \ldots, \mathbf{l}_n \in \mathbf{A}$:*

   $$[\delta(\mathbf{f})]^3_{\mathbf{M}}\theta' = \mathbf{true} \quad \textit{if and only if} \quad \langle \mathbf{l}_1, \ldots, \mathbf{l}_n \rangle \in \mathbf{dom}(\mathbf{I}(\mathbf{f}))$$

   where $\theta' = [v_1 \to \mathbf{l}_1, \ldots, v_n \to \mathbf{l}_n]$ and $\{v_1, \ldots, v_n\}$ are the parameter names of $\mathbf{f}$.

2. *The domain formulas of δ are well-defined for variable assignment θ. That is, for each $\mathbf{f} \in \mathbf{F}$: $\mathbf{wd}([\delta(\mathbf{f})]^3_{\mathbf{M}}\theta) = \mathbf{true}$.*

Remark. Note that the definition only mentions functions in $\mathbf{F}$, and does not mention predicates in $\mathbf{P}$. This is because approaches in the literature assume that predicates are total on well-defined arguments.

The $\mathcal{D}$ operator was developed for the standard syntax of first-order logic, and the interpretation of logical connectives follows the interpretation of Strong Kleene [71]. For instance, as the truth table in Figure 5.1(a) shows, a conjunction is well-defined if and only if either (1) both conjuncts are well-defined, or (2) if one of the conjuncts is well-defined and evaluates to **false**. Intuitively, in case (1) the classical two-valued evaluation can be applied, while in case (2) the truth value of the conjunction is **false** independently of the well-definedness and value of the other conjunct. The interpretation of universal quantification is the generalization of conjunction: it is well-defined

[1] The literature leaves domain restriction δ an implicit argument of the operator. We keep that notation, too.

[2] Function **wd** was defined in Section 2.3 on page 26.

$\wedge$	**true**	**false**	$\bot$
true	**true**	**false**	$\bot$
false	**false**	**false**	**false**
$\bot$	$\bot$	**false**	$\bot$

(a) Strong Kleene

$\wedge$	**true**	**false**	$\bot$
true	**true**	**false**	$\bot$
false	**false**	**false**	**false**
$\bot$	$\bot$	$\bot$	$\bot$

(b) McCarthy

Figure 5.1: Kleene's and McCarthy's interpretation of conjunction

if and only if either (1) the quantified expression is well-defined for every instantiation of the quantified variable, or (2) the quantified expression is well-defined and yields **false** for some instantiation of the quantified variable.

The $\mathcal{D}$ operator proved to be impractical: the size of well-definedness conditions it generates is exponential with respect to the size of the input formula [117, 4]. Thus, instead of $\mathcal{D}$, another formula transformer $\mathcal{L}$ was proposed, which generates much smaller conditions with linear growth, but which is *incomplete*. That is, the $\mathcal{L}$ operator is stronger than $\mathcal{D}$: $\mathcal{L}(\varphi) \Rightarrow \mathcal{D}(\varphi)$ holds, but $\mathcal{D}(\varphi) \Rightarrow \mathcal{L}(\varphi)$ does not necessarily hold [16]. This means that the $\mathcal{L}$ operator may generate unprovable well-definedness conditions for well-defined formulas. Despite this drawback, due to the significantly smaller well-definedness conditions that $\mathcal{L}$ generates, many tools use the $\mathcal{L}$ operator [117, 16].

Remark. Work on the proof theory of many-valued logics lead to the development of operators that are equivalent to the $\mathcal{D}$ operator and do not cause the exponential explosion of well-definedness conditions [53, 54]. However, these results were mostly not picked up by the communities working on design languages and software verification.

The main difference between the $\mathcal{D}$ and $\mathcal{L}$ operators is that they use slightly different interpretations of connectives and quantifiers. For the interpretation of connectives, the interpretation of McCarthy is used [92]. As an example, the truth table in Figure 5.1(b) presents McCarthy's interpretation of conjunction. The only difference from Kleene's interpretation is in the interpretation of $\bot \wedge$ **false**, which yields $\bot$. This reveals the most important difference between the two interpretations: in McCarthy's interpretation conjunction and disjunction are not commutative. This is because McCarthy's interpretation is *sequential* [55], that is, the order of the operands might make a difference in the definedness of an expression.

The difference in the interpretation of quantifiers is that $\mathcal{L}$ considers a quantification to be defined if and only if the quantified expression is well-defined for all possible instantiations of the quantified variable. This interpretation of quantifiers is commonly referred to as the *strict* interpretation [55].

In the rest of the thesis, the $\mathcal{L}$ operator will be used for the well-definedness checking of logical formulas. Therefore, the formal details of the $\mathcal{L}$ operator are summarized in Appendix A. Figure A.2 on page 184 presents the three-valued interpretation of first-order terms and formulas to which the well-definedness conditions that $\mathcal{L}$ yields correspond. The definition of $\mathcal{L}$ is given in Figure A.1.

An important result for our purposes is the following theorem, which expresses that for McCarthy's interpretation (presented in Figure A.2) $\mathcal{L}$ is a syntactical characterization of the semantical operation **wd** [16, 17].[3]

Theorem 5.1. *For every structure* $\mathbf{M}$*, domain restriction* δ*, formula* φ*, and variable assignment* θ*, the following holds:*

$$(\mathbf{M},\theta)^{\delta} \;\Rightarrow\; (\,[\mathcal{L}(\varphi)]^{2}_{\widehat{\mathbf{M}}}\theta = \mathbf{wd}([\varphi]^{3}_{\mathbf{M}}\theta)\,)$$

Recall that $\widehat{\mathbf{M}}$ is a total structure, the extension of structure $\mathbf{M}$. Consequently, the theorem expresses that the $\mathcal{L}$ operator allows one to check the well-definedness of three-valued formulas in two-valued logic.

Remark. As mentioned above, for Kleene's interpretation $\mathcal{D}$ characterizes operation **wd**, while $\mathcal{L}$ is stronger than $\mathcal{D}$. Thus, for Kleene's interpretation the relation between $\mathcal{L}$ and **wd** is: $(\mathbf{M},\theta)^{\delta} \Rightarrow (\,[\mathcal{L}(\varphi)]^{2}_{\widehat{\mathbf{M}}}\theta \Rightarrow \mathbf{wd}([\varphi]^{3}_{\mathbf{M}}\theta)\,)$.

5.2 Well-definedness of JML^{--} Expressions

The operators described above are applicable to first-order formulas. In this section, we show how these results can be lifted to specification expressions. In particular, we introduce operator *Df* that generates well-definedness conditions for JML^{--} expressions.

The main decision to make when designing *Df* is what the interpretation of logical operators should be. The interpretation imposed by operator $\mathcal{D}$, operator $\mathcal{L}$, a mixture of the two, or something completely different?

Since Java's evaluation of expressions is sequential and is from left to right, the semantics of the conditional boolean operators coincides with McCarthy's interpretation of logical connectives. For instance, McCarthy's interpretation of conjunction in Figure 5.1(b) is in line with Java's `&&` operator.

The semantics of JML does not explicitly specify the conditions under which a quantification is considered to be well-defined. That is, it is not clear whether quantifiers have a strict interpretation or not. Therefore, we are free to choose one or the other. We prefer a strict interpretation because if a quantified expression is well-defined for all instantiations, then

[3] Behm *et al.* [16] assume domain restrictions to contain only total operators, therefore premise $(\mathbf{M},\theta)^{\delta}$ is left out in their formalization as it is always trivially true.

the interpretation of the quantification and the result of runtime assertion checking surely match.

These considerations indicate that not only generates operator $\mathcal{L}$ smaller well-definedness conditions than $\mathcal{D}$, the interpretation that $\mathcal{L}$ leads to is also closer to the expectations of programmers than the interpretation resulting from the use of $\mathcal{D}$. Thus, we build on the $\mathcal{L}$ operator and lift it to the syntax of JML^{--} expressions to obtain the *Df* operator.

Remark. The definition of both $\mathcal{D}$ and $\mathcal{L}$ corresponds to a *strict* interpretation of function and predicate symbols. That is, an undefined argument makes their interpretation to be undefined. This coincides with the semantics of operators and method calls in Java where the evaluation of all operands or actual parameters has to succeed in order to evaluate an operator or call, respectively.

Definition of *Df* and its Relation to $\mathcal{L}$. Next, we define the *Df* operator and show that it is an instance of operator $\mathcal{L}$ with a domain restriction that directly corresponds to the semantics of the partial operations of JML^{--}expressions.

The signature of the *Df* operator is similar to that of the encoding function γ, except that *Df* always yields a first-order formula:

$$Df : Expr \times \textbf{Store} \times \textbf{Store} \rightarrow Formula$$

The two store arguments denote the post- and prestore, respectively.

The definition of the *Df* operator is given in Figure 5.2. Well-definedness of conjunction, disjunction, and implication is defined according to the sequential evaluation described above. That is, they are well-defined if and only if either both operands are well-defined or the first operand is defined and the second operand does not need to be evaluated for the evaluation of the whole expression.

The symbol $\oplus$ denotes any of the JML^{--} operators `==`, `!=`, `<`, `<=`, `+`, `-`, and `*`. These operators are total on well-defined operands, thus they are well-defined if and only if both operands are well-defined. The symbol $\otimes$ denotes any of the operators `/` and `%`. These operators are partial since their domains exclude 0 being the second argument. Thus, they are well-defined if and only if both operands are well-defined and the second operand is different from 0. Negation is well-defined if and only if the operand is well-defined.

A field access is well-defined if and only if the expression that denotes the target object is well-defined and evaluates to a value other than **`null`**. A call to some pure method `m` in type `T` with formal parameters $p_1, \ldots, p_n$ is well-defined if and only if (1) the expression that denotes the receiver object is well-defined and evaluates to some value other than **`null`**; (2) all parameters are well-defined; and (3) the precondition $\mathrm{pre}^{\mathtt{T}}_{\mathtt{m}}$ of the method

$$
\begin{array}{lcl}
Df(E\ \texttt{\&\&}\ F, OS', OS) & \triangleq & Df(E, OS', OS) \wedge (\gamma(E, OS', OS) \Rightarrow Df(F, OS', OS)) \\
Df(E\ \texttt{||}\ F, OS', OS) & \triangleq & Df(E, OS', OS) \wedge (\neg\gamma(E, OS', OS) \Rightarrow Df(F, OS', OS)) \\
Df(E\ \texttt{==>}\ F, OS', OS) & \triangleq & Df(E, OS', OS) \wedge (\gamma(E, OS', OS) \Rightarrow Df(F, OS', OS)) \\
Df(E \oplus F, OS', OS) & \triangleq & Df(E, OS', OS) \wedge Df(F, OS', OS) \\
Df(E \otimes F, OS', OS) & \triangleq & Df(E, OS', OS) \wedge Df(F, OS', OS) \wedge \gamma(F, OS', OS) \neq 0 \\
Df(\texttt{!}E, OS', OS) & \triangleq & Df(E, OS', OS) \\
Df(E\texttt{.f}, OS', OS) & \triangleq & Df(E, OS', OS) \ \wedge\ \gamma(E, OS', OS) \neq null \\
Df(E\texttt{.m(}e_1, ..., e_n\texttt{)}, OS', OS) & \triangleq & \\
\multicolumn{3}{l}{\quad Df(E, OS', OS) \wedge \bigwedge_{i=1}^{n} Df(e_i, OS', OS) \ \wedge \gamma(E, OS', OS) \neq null \wedge} \\
\multicolumn{3}{l}{\quad Df(\mathrm{pre}^{\mathtt{T}}_{\mathtt{m}}[E/\texttt{this}, e_1/p_1, ..., e_n/p_n], OS', OS) \wedge} \\
\multicolumn{3}{l}{\quad \gamma(\mathrm{pre}^{\mathtt{T}}_{\mathtt{m}}[E/\texttt{this}, e_1/p_1, ..., e_n/p_n], OS', \_)} \\
Df(\texttt{new C(}e_1, ..., e_n\texttt{)}, OS', OS) & \triangleq & \\
\multicolumn{3}{l}{\quad \bigwedge_{i=1}^{n} Df(e_i, OS', OS) \ \wedge\ Df(\mathrm{pre}_{\mathtt{C}}[e_1/p_1, ..., e_n/p_n], OS', OS) \wedge} \\
\multicolumn{3}{l}{\quad \gamma(\mathrm{pre}_{\mathtt{C}}[e_1/p_1, ..., e_n/p_n], OS', \_)} \\
Df(\mathtt{v}, OS', OS) & \triangleq & true \\
Df(\texttt{\textbackslash old(}E\texttt{)}, OS', OS) & \triangleq & Df(E, OS, \_) \\
Df(\texttt{\textbackslash fresh(}E\texttt{)}, OS', OS) & \triangleq & Df(E, OS', OS) \\
Df(\texttt{(\textbackslash forall }T\ x\texttt{. }E\texttt{)}, OS', OS) & \triangleq & \forall x.\ (allocT(x, OS', T) \Rightarrow Df(E, OS', OS)) \\
Df(\texttt{(\textbackslash exists }T\ x\texttt{. }E\texttt{)}, OS', OS) & \triangleq & \forall x.\ (allocT(x, OS', T) \Rightarrow Df(E, OS', OS))
\end{array}
$$

Figure 5.2: Definition of the Df operator

is well-defined and holds. The well-definedness of constructors is defined analogously.

Parameter names and literals are always well-defined. A fresh-expression is well-defined if and only if its argument is well-defined. An old-expression is well-defined if and only if its argument is well-defined in the prestore.

Quantifiers are well-defined if and only if the quantified expression is well-defined for every instantiation of the quantified variable. As defined in Section 2.1, quantification over reference-type variables ranges over allocated objects of the proper type. Thus, the quantified expression has to be well-defined over this range of objects. In Figure 5.2, the shaded sub-formula is only added for quantifiers over reference types.

Note that, for instance, a field access or a method call on a non-allocated object yields an ill-defined expression. This is one of the reasons why the semantics of JML^{--} restricts the scope of quantifiers with reference-type bound variables to range over allocated objects only.

Remark. Note that the aliveness of accessed variables and locations does not have to be shown when proving the well-definedness of an expression. This is because, for most cases, the language ensures that this property holds.

The only two exceptional cases are `\result` and quantified variables. For `\result`, the property holds due to the restriction we made on its usage in old-expressions (see Section 2.1 on page 22). For quantified variables, the property follows from the semantics defined for variables of reference types.

Relating *Df* and $\mathcal{L}$. We show that *Df* is an instance of $\mathcal{L}$ by formally proving the following theorem.

Theorem 5.2. *For every* JML⁻⁻ *expression* E*, and stores* OS' *and* OS*, the following equation holds:* $Df(E, OS', OS) \equiv \mathcal{L}(\gamma(E, OS', OS))$.

Before the proof could be presented, there are two points to consider.

Relating Syntax. On the right-hand side of the equation, $\mathcal{L}$ is applied on terms and formulas that γ yields. In order to know how $\mathcal{L}$ is to be applied, we need to relate the syntactic elements that appear in such terms and formulas with the syntactic elements of first-order logic. For instance, to determine which rule of $\mathcal{L}$ is to be applied on formula $\mathcal{L}(x \leq y)$, we need to know that symbol $\leq$ corresponds to a predicate.

To find all syntactic elements that may appear in encodings of JML⁻⁻ expressions, the definition of γ in Figure 4.3 on page 50 has to be inspected. The relation, presented in Figure 5.3(a), is obtained in a straightforward manner. Note that if function *loc* is considered to be partial then the lookup function $\_(\_)$ is total. Symbols $\widehat{m_T}$ and $\widehat{C}$ represent the function symbols that are introduced for pure methods and constructors, respectively. Constants *true* and *false*, as well as the logical connectives are omitted as they are part of the syntax of first-order logic themselves.

Domain Restriction. Although implicit in the notation, some domain restriction is always an argument of $\mathcal{L}$. Of course, not an arbitrary one: in this case, one that reflects the semantics of Java and JML. Therefore, we have to define the domain restrictions of the five partial function symbols identified in Figure 5.3(a). The domain restrictions, presented in Figure 5.3(b),[4] are rather straightforward, and will be denoted by δ_γ in the sequel.

The proof of Theorem 5.2 is presented in Appendix B.

Preserving Semantics of Expressions. One of the concerns with the encoding of specification expressions was that the original semantics of expressions should be preserved (see Section 3.1.1). Based on (1) the interpretation presented in Figure A.2 on page 184, which coincides with the semantics of Java and which is the basis of the definition of operator $\mathcal{L}$, and (2) Theorem 5.2 that expresses the relation between $\mathcal{L}$, γ, and *Df*; we can

[4] For simplicity, we assume that o and p are literals or parameter names. Otherwise, the substitutions would require a more complicated formula.

symbols in γ-encodings	FOL counterparts
$=, \neq, <, \leq$	predicates
$+, -, *$	total functions
div, mod, loc, $\widehat{m_T}$, $\widehat{C}$	partial functions
$\_(\_)$ (lookup), $alive$, $alloc$, $allocT$	total functions
$null$, -1, 0, 1, ...	total (constant) functions
$this$, $resV$, $param$, f (field), OS (store), T (type), x (bound variable)	variables

(a)

partial functions	domain restrictions
$div(x, y)$	$y \neq 0$
$mod(x, y)$	$y \neq 0$
$loc(o, f)$	$o \neq null$
$\widehat{m_T}(o, p, OS')$	$o \neq null \ \wedge \ Df(\mathrm{pre}^{\mathtt{T}}_{\mathtt{m}}[o/\mathtt{this}, p/\mathtt{param}], OS', OS) \wedge$ $\gamma(\mathrm{pre}^{\mathtt{T}}_{\mathtt{m}}[o/\mathtt{this}, p/\mathtt{param}], OS', \_)$
$\widehat{C}(p, OS')$	$Df(\mathrm{pre}_{\mathtt{C}}[p/\mathtt{param}], OS', OS) \wedge$ $\gamma(\mathrm{pre}_{\mathtt{C}}[p/\mathtt{param}], OS', \_)$

(b)

Figure 5.3: Relating symbols and defining their domain restrictions

now conclude that the use of Df and γ *together* ensure that the semantics of expressions is preserved.

Proving Well-definedness Conditions. The definition of Df provides a means to generate conditions for JML^{--} expressions whose validity ensures that the expressions are well-defined. However, it is not yet clarified what may be assumed when attempting to prove given well-definedness conditions.

Recall from Example 3.5 on page 35 that specification elements may depend on each other. For instance, the well-definedness of a specification element may only be provable if other specification elements and axioms generated over certain function symbols can be used. On the other hand, to prevent using mathematically unreasonable specifications and axioms in proofs, only well-defined specifications should be assumed and axiomatized.

This suggests that (1) dependencies between specification elements have to be taken into account, and that (2) the well-definedness checking of specifications and the axiomatization of pure methods mutually depend on each other, and hence, are to be done simultaneously.

The development of a technique that provides solutions to these two points is the topic of the next chapter.

Remark. Related work on the well-definedness checking of specification expressions is discussed in the next chapter, where our technique of proving well-definedness conditions is presented.

Chapter 6

Axiomatization of Pure Methods

In the previous chapters, we have seen how specification expressions can be encoded by function γ in the programming logic presented in Section 2.2, and how well-definedness conditions for such specification expressions can be generated by operator Df. And, we have seen that encodings of specification elements and well-definedness conditions may contain applications of uninterpreted function symbols.

Reasoning about such formulas is only possible if meaning is assigned to these function symbols. The standard way to do so, is to axiomatize them based on user-defined method specifications and invariants [32, 35, 33]. The main challenge of axiomatizing specifications is to ensure consistency of the resulting axiom system. Clearly, this is crucial to every formal approach; otherwise unsound reasoning may occur.

This chapter is based on our previous work [116]. We describe a technique for the axiomatization of pure methods. The technique ensures *well-formedness* of the specification on which the axiomatization is based. A specification is called well-formed if it is both *well-defined* and *consistent*.

A specification is considered well-defined if all its specification elements are well-defined in the sense described in the previous chapter. A specification is considered consistent if the axiom system that is extracted from it for pure methods is free of contradictions.

The basis of the presented technique is a set of criteria that precisely define when a specification is well-formed. These criteria are enforced by imposing proof obligations on specification elements that ensure: (1) their well-definedness, (2) the existence of a possible result value for each pure method, and (3) that recursive specifications are well-founded. Our soundness result states that if all proof obligations are provable, then the specification fulfills the criteria.

In order to deal with dependencies between pure methods, we determine

a *dependency graph*, which we process bottom-up. Thereby, one can use the properties of a method m to prove the proof obligations for the methods using m.

The proposed technique checks the well-formedness of specifications independently of implementations, because an inconsistent method specification is not necessarily detected during source-code verification for the following reasons [35]: (1) methods may not have implementations (because of being abstract or being declared in an interface); (2) partial correctness logics allow one to verify non-terminating implementations with respect to unsatisfiable specifications; (3) any implementation could be trivially verified based on inconsistent axioms stemming from inconsistent pure-method specifications; this is especially true for recursive implementations, where the specification of the method is needed to verify the implementation.

Remark. The axioms that the presented technique captures reflect the behavior of pure methods as described by their specifications. Our technique does not cover other kind of axioms, in particular, axioms that describe on which parts of the heap does the return value of a pure method potentially depend. We refer the reader for axioms of that kind to the approaches of inspector methods [65], confined methods in Spec# [33], dynamic frames [128, 127], equivalent-results methods [85], and the use of location descriptors [25]. These papers argue for the soundness of additionally introduced axioms.

6.1 Formalization of Specification Elements

We assume a set of function symbols $\mathbf{F} \triangleq \{f_1,\ f_2, , \ldots,\ f_n\}$, which corresponds to the set of pure methods of a program.[1] For simplicity, we assume that pure methods have exactly one explicit parameter p. Thus, all functions in $\mathbf{F}$ are ternary with parameters for the receiver object (`this`), the explicit parameter (p), and the store (OS).

Remark. The simplifying assumption above rules out pure constructors, which do not have a receiver object in our formalization. However, their handling is analogous to instance pure methods.

We define a specification of $\mathbf{F}$ as $\mathbf{Spec} \triangleq \langle \mathbf{Pre}, \mathbf{Post}, \mathbf{INV} \rangle$, where:

• $\mathbf{Pre}$ maps each $f \in \mathbf{F}$ to a formula. $\mathbf{Pre}(f)$ is the encoding of precondition pre_f of method f:

$$\mathbf{Pre}(f) \ \triangleq\ o \neq \mathit{null} \wedge \gamma(\mathrm{pre}_f[o/\texttt{this}], OS, \_)$$

[1] For simplicity, even pure methods of boolean type are treated as functions and not as predicates.

Due to the definition and the syntactic structure of preconditions, the only free variables in $\mathbf{Pre}(f)$ are OS, o, and p.

The first conjunct expresses that the receiver object is different from `null`. This property does not need to be added to user-declared preconditions as the semantics of the programming language ensures that the object referred to by `this` is never `null`. Still, we conjoin the property to $\mathbf{Pre}(f)$ because this will allow us to use **Pre** as domain restriction as explained in Section 6.4.2.

Remark. For simplicity, we assume that the use of identifiers o, p, and OS in the definitions does not lead to the confusion and capturing of variable and field names.

• **Post** maps each $f \in \mathbf{F}$ to a formula. $\mathbf{Post}(f)$ is the encoding of postcondition post_f of method f:

$$\mathbf{Post}(f) \triangleq \gamma(\text{post}_f[o/\texttt{this}], OS, OS)$$

Due to the above definition and the syntactic structure of postconditions, the only free variables in $\mathbf{Post}(f)$ are OS, o, p, and the result variable $resV$. Due to the simplified encoding of pure-method specifications, the same store variable is used to denote the pre- and poststore of pure methods.

• **INV** is a set of formulas $\{\mathbf{Inv}_1, \mathbf{Inv}_2, \ldots, \mathbf{Inv}_m\}$, where $\mathbf{Inv}_i$ is the encoding of the i-th invariant Inv_i of the program. Assuming that Inv_i is declared in type T, $\mathbf{Inv}_i$ is defined as:

$$\mathbf{Inv}_i \triangleq \mathit{typeof}(o) \preceq T \Rightarrow \gamma(Inv_i[o/\texttt{this}], OS, \_)$$

The left-hand side of the implication makes sure that invariant Inv_i is only related with objects of the appropriate type. Due to the above definition and the syntactic structure of invariants, the only free variables in $\mathbf{Inv}_i \in \mathbf{INV}$ are OS and o.

In order to simplify formulas in the sequel, we use **SysInv** to denote the conjunction of all invariants for all allocated objects:

$$\mathbf{SysInv} \triangleq \forall\, o.\ \mathit{alloc}(o, OS) \Rightarrow \bigwedge_{i=1}^{m} \mathbf{Inv}_i$$

Note that **SysInv** is an open formula with free variable OS, and that predicate *alloc* "filters out" the `null` value.

Normal Behavior. As described in Section 2.1, a method specification in JML^{--} only describes *normal behavior*, that is, behavior in which the

corresponding method does not throw exceptions. Therefore, **Pre** and **Post** only contains formulas extracted from normal-behavior specifications.

Specifications that prescribe behavior when methods must or may throw exceptions (*exceptional behavior* and *behavior* specification cases in JML) are not of concern for our approach because the kind of axioms we are interested in capture information on the return values of pure methods. Pure methods that throw exceptions have no return values.

In fact, our approach considers calls that lead to exceptions to be ill-defined: generated well-definedness conditions require that preconditions of the (implicitly) normal behavior method specifications hold.

6.2 Well-formedness Criteria

In this section, we define the four criteria that a program specification has to fulfill to be considered well-formed.

Recall that one of the two components of well-formedness is consistency. The literature (*e.g.*, [125, 41, 106]) proposes two main approaches to ensure that a specification is consistent: (1) the exclusive use of *conservative extensions*, and (2) the exhibition of a *model* for the specification.

Conservative extensions are commonly used in theorem provers where syntactic restrictions guarantee that definitions are indeed conservative extensions [106]. Unfortunately, this approach is not an option in our setting because interface specifications are typically axiomatic, and therefore not conservative extensions.

The approach of exhibiting a model for the specification fits well with the axiomatic nature of interface specifications, and is the approach that we apply to show consistency of specifications. Consequently, the criteria of well-formedness below are expressed in terms of structures.

Structures and Interpretation. To define the interpretation of specifications, we use a structure $\mathbf{M} \triangleq \langle \mathbf{Value}, \mathbf{Store}, \mathbf{I} \rangle$, where $\mathbf{I}$ is an interpretation function for the functions in $\mathbf{F}$. For some function $f \in \mathbf{F}$, the arguments of $\mathbf{I}(f)$ are the same as the arguments of f:

$$\mathbf{I}(f) : \mathbf{Value} \times \mathbf{Value} \times \mathbf{Store} \to \mathbf{Value}$$

Well-formedness Criteria. The criteria of well-formedness are expressed in terms of (partial) structures, function **wd** (defined in Section 2.3 on page 26), and the specification elements of the program at hand. The four criteria are presented on the next page.

1. Invariants are never interpreted as $\perp$, that is, for each $\mathbf{store} \in \mathbf{Store}$:

   $\mathbf{wd}([\mathbf{SysInv}]^3_{\mathbf{M}}\theta)$ holds

   where $\theta \triangleq [OS \to \mathbf{store}]$ is a variable assignment.

2. Preconditions are never interpreted as $\perp$ in stores that satisfy the invariants of all allocated objects, that is, for each $f \in \mathbf{F}$, $\mathbf{store} \in \mathbf{Store}$, $\mathbf{this} \in \mathbf{Value}$, and $\mathbf{par} \in \mathbf{Value}$:

   if $[\mathbf{SysInv}]^3_{\mathbf{M}}\theta$ holds, and
   $\mathbf{this}$ is an allocated object of type T and $\mathbf{par}$ is alive in $\mathbf{store}$,
   then $\mathbf{wd}([\mathbf{Pre}(f)]^3_{\mathbf{M}}\theta)$ holds

   where $\theta \triangleq [OS \to \mathbf{store}, o \to \mathbf{this}, p \to \mathbf{par}]$, and T is the type in which the method that corresponds to f is declared.

3. The values of the parameters belong to the domain of the interpretation of function symbols, provided that the store satisfies the invariants and the precondition holds. That is, for each $f \in \mathbf{F}$, $\mathbf{store} \in \mathbf{Store}$, $\mathbf{this} \in \mathbf{Value}$, and $\mathbf{par} \in \mathbf{Value}$:

   if $[\mathbf{SysInv}]^3_{\mathbf{M}}\theta$ holds, and
   $\mathbf{this}$ is an allocated object of type T and $\mathbf{par}$ is alive in $\mathbf{store}$,
   and $[\mathbf{Pre}(f)]^3_{\mathbf{M}}\theta$ holds,
   then $\langle \mathbf{this}, \mathbf{par}, \mathbf{store} \rangle \in \mathbf{dom}(\mathbf{I}(f))$ holds

   where $\theta \triangleq [OS \to \mathbf{store}, o \to \mathbf{this}, p \to \mathbf{par}]$, and T is the type in which the method that corresponds to f is declared.

4. Postconditions are never interpreted as $\perp$ for any result, and the interpretation of function f as result value satisfies the postcondition, provided that the store satisfies the invariants and the precondition holds. That is, for each $f \in \mathbf{F}$, $\mathbf{store} \in \mathbf{Store}$, $\mathbf{this} \in \mathbf{Value}$, and $\mathbf{par} \in \mathbf{Value}$:

   if $[\mathbf{SysInv}]^3_{\mathbf{M}}\theta$ holds, and
   $\mathbf{this}$ is an allocated object of type T and $\mathbf{par}$ is alive in $\mathbf{store}$,
   and $[\mathbf{Pre}(f)]^3_{\mathbf{M}}\theta$ holds,
   then
   for each $\mathbf{result} \in \mathbf{Value}$ alive in $\mathbf{store}$: $\mathbf{wd}([\mathbf{Post}(f)]^3_{\mathbf{M}}\theta')$ holds,
   and $[\mathbf{Post}(f)]^3_{\mathbf{M}}\theta$ holds

   where $\theta' \triangleq [OS \to \mathbf{store}, o \to \mathbf{this}, p \to \mathbf{par}, resV \to \mathbf{result}]$ and
   $\theta \triangleq [OS \to \mathbf{store}, o \to \mathbf{this}, p \to \mathbf{par}, resV \to \mathbf{I}(f)(\mathbf{this}, \mathbf{par}, \mathbf{store})]$,
   and T is the type in which the method that corresponds to f is declared.

   Note that in θ, variable $resV$ is assigned the value of the interpretation of function f with parameters $\mathbf{this}$, $\mathbf{par}$, and $\mathbf{store}$.

Based on the four criteria, we can define partial models for and well-formedness of specifications as follows.

Definition 6.1. ***(Partial model for specification)*** *A structure* $\mathbf{M}$ *is a* partial model *for specification* **Spec**, *denoted by* $wf(\mathbf{Spec}, \mathbf{M})$, *if it satisfies all four criteria.*

Definition 6.2. ***(Well-formed specification)*** *A specification* **Spec** *is well-formed, denoted by* $wf(\mathbf{Spec})$, *if there exists a partial model* $\mathbf{M}$ *for the specification.*

6.3 Axiomatization

Before showing how our approach ensures that a specification fulfills the four criteria, we present the axioms that are extracted from the specification in case it is well-formed.

The axiom extracted for symbol f_T that corresponds to pure method f in type T is denoted by $\mathbf{Ax}_{f_T}$ and is defined as follows:

$$\begin{aligned}\mathbf{Ax}_{f_T} \triangleq\ & \forall\ OS, o, p. \\ & \mathbf{SysInv} \wedge \mathit{allocT}(o, OS, T) \wedge \mathit{alive}(p, OS) \wedge \mathbf{Pre}(f_T) \Rightarrow \\ & \mathit{alive}(f_T(o, p, OS), OS) \wedge \mathbf{Post}(f_T)[f_T(o, p, OS)/\mathit{resV}]\end{aligned} \tag{6.1}$$

Remark. According to the visible-state invariant semantics, **SysInv** should be conjoined to the consequence of the implication, too. However, that would just be a tautology in the consequence: the same store OS is used on both sides of the implication, **SysInv**'s only free variable is OS, and **SysInv** is present in the premise.

To simplify formalizations, we define $\mathbf{Ax}_{f_T} \triangleq \mathit{true}$ if type T inherits method f and therefore does not contain a method specification for f.

Recall that function γ encodes a method call according to the static type of the receiver object. For instance, a call to method f with a receiver of static type T is encoded by an application of function symbol f_T.

In order to model inheritance and dynamic method binding, we need to make the connection between the uninterpreted function symbol that models method f in type T and the symbol that models f in T's direct subtype S. For this purpose, the following axiom is extracted if f is inherited by S or if f is an overriding method in S:

$$\mathbf{Ax}_{f_{S\preceq}} \triangleq\ \forall\ OS, o, p.\ \mathit{typeof}(o) \preceq S\ \Rightarrow\ f_S(o, p, OS) = f_T(o, p, OS)$$

To simplify formalizations, we define $\mathbf{Ax}_{f_{S\preceq}} \triangleq \mathit{true}$ if method f is introduced by S.

The axiom $\mathbf{Ax}_{f_{S\preceq}}$ helps in two ways when reasoning about pure method calls. First, assume that the γ-encoding of a specification expression contains an application of symbol f_T and the dynamic type of the receiver is known to be S, which overrides method f. Then the axiom tells that the result of function f_T is the same as that of f_S, thereby the properties expressed by axiom $\mathbf{Ax}_{f_S}$ can also be used to reason about the application of symbol f_T.

Second, assume that the γ-encoding of a specification expression contains an application of symbol f_S and method f is inherited by S from T. Since S contains no method specification for f, there is no axiom $\mathbf{Ax}_{f_S}$ extracted for symbol f_S. However, $\mathbf{Ax}_{f_{S\preceq}}$ tells that the result of function f_T is the same as that of f_S, thereby the properties expressed by axiom $\mathbf{Ax}_{f_T}$ can also be used to reason about the application of symbol f_S.

We denote the axioms for all function symbols in $\mathbf{F}$ by $\mathbf{Ax_{Spec}}$. Formally:

$$\mathbf{Ax_{Spec}} \triangleq \bigwedge_{f_T \in \mathbf{F}} \mathbf{Ax}_{f_T} \ \wedge \ \mathbf{Ax}_{f_{T\preceq}}$$

Crucial for our approach is that the axiom system $\mathbf{Ax_{Spec}}$ is consistent. The following theorem states that this is the case if the specification from which $\mathbf{Ax_{Spec}}$ was extracted is well-formed.

Theorem 6.1. ***(Consistency of axioms)*** *If specification* **Spec** *is well-formed then* $\mathbf{Ax_{Spec}}$ *is consistent.*

Proof. We prove the consistency of $\mathbf{Ax_{Spec}}$ by showing that there is a structure that is a model for $\mathbf{Ax_{Spec}}$. From the definition of $\mathbf{Ax_{Spec}}$, we can see that this can be shown by proving that there is a structure that is a model for axiom $\mathbf{Ax}_{f_T}$ and axiom $\mathbf{Ax}_{f_{T\preceq}}$ for all function symbols f_T in $\mathbf{F}$. We prove the existence of such a structure for the two kinds of axiom separately, and begin with $\mathbf{Ax}_{f_T}$.

By assumption, **Spec** is well-formed. Therefore, by definition (Definitions 6.2 and 6.1), we know that there exists some structure $\mathbf{M}$ that is a partial model for **Spec** and that fulfills all four criteria presented in the previous section.

Axiom $\mathbf{Ax}_{f_T}$ captures the semantic property expressed in Criterion 4 over function f in type T by the means of the programming logic. Therefore, we can deduce that $\mathbf{M}$ is also a partial model for axiom $\mathbf{Ax}_{f_T}$. The same argument holds for all other function symbols in $\mathbf{F}$.

This concludes the proof for the first kind of axiom. It remains to prove that $\mathbf{M}$ is also a model for axiom $\mathbf{Ax}_{f_{T\preceq}}$ for all function symbols f_T in $\mathbf{F}$. This follows from behavioral subtyping (see Assumption 2.3 on page 21), which ensures that the axiom for an overriding method is stronger than the

axiom for the overridden method. Thus, if $\mathbf{M}$ is a partial model for axiom $\mathbf{Ax}_{f_T}$ for all symbols $f_T \in \mathbf{F}$, then $\mathbf{M}$ is also a partial model for axiom $\mathbf{Ax}_{f_{T\preceq}}$ for all symbols $f_T \in \mathbf{F}$.

Thereby, we have proved that $\mathbf{M}$ is a partial model for $\mathbf{Ax}_{\mathbf{Spec}}$, which concludes the proof. □

Important to note is that this property does not hold in the other direction, that is, it is not necessarily true that if $\mathbf{Ax}_{\mathbf{Spec}}$ is consistent then specification **Spec** is well-formed. For example, consider a method with precondition `1/0 == 1/0` and postcondition **`true`**. In two-valued logic, the resulting axiom would be trivially consistent, but the specification is not well-formed (Criterion 2). This demonstrates that our well-formedness criteria require more than just consistency, namely also satisfaction of partiality constraints.

Remark. The same way as with symbol names, we will drop the subscripts denoting types in the names of axioms whenever they are not relevant. For instance, we will use $\mathbf{Ax}_f$ instead of $\mathbf{Ax}_{f_T}$.

6.3.1 Feasibility of Invariants

Note that the four criteria only required the well-definedness of invariants, but not their feasibility. Therefore, for instance, the infeasible invariant "**`this`**`.f > 0 &&` **`this`**`.f < 0`" satisfies the criteria, in particular Criterion 1), because the expression is well-defined.

We permit infeasible invariants for two reasons. First, axiom system $\mathbf{Ax}_{\mathbf{Spec}}$ remains sound even in the presence of infeasible invariants. This is due to the shape of generated axioms: **SysInv** is always conjoined to the premise of implications (see formula (6.1) above). Thus, infeasible invariants make **SysInv** to evaluate to **false**, thereby making the axioms void.

Second, in the presence of infeasible invariants, it is not possible to construct valid objects. This problem is caught when verifying the implementations of constructors—unless a partial correctness logic is used and the constructor does not terminate.

With additional proof effort the feasibility of invariants could be checked, too. We do not check this property because it is orthogonal to our aim of generating consistent axiom systems.

6.3.2 Method Calls in Invariants

The use of pure-method calls is just as natural in invariants as in method specifications. For instance, in Figure 3.3 on page 36, two of the three invariants contained method calls. However, such calls lead to serious incompleteness issues as illustrated by the following example.

```
class Incomplete {
  int f=1;
  invariant positive();
  // constraint \old(this.f) <= this.f;

  int increment()
    ensures this.f == \old(this.f) + 1;
    // assignable this.f;
  { this.f++; }

  pure boolean positive()
    ensures result == this.f > 0;
    // reads this.f;
  { return this.f > 0; }
}
```

Figure 6.1: Incompleteness due to method call in invariant

Example 6.1. Class `Incomplete` in Figure 6.1 contains an integer field `f`, a mutating method that increments the value of the field by 1, and a pure method that returns **true** if and only if the value of the field is greater than 0.[2] The invariant of the class contains a method call expressing that in every visible state method `positive` should return **true**.

It is easy to see that the specification of the class is well-formed. Axiom $\mathbf{Ax}_{\widehat{positive}}$ that is generated for pure method `positive` is the following:

$$\begin{array}{l}\forall\ OS, o.\\ \quad (\forall\ o.\ alloc(o, OS)\ \Rightarrow\ typeof(o) \preceq \texttt{Incomplete}\ \Rightarrow\ \widehat{positive}(o, OS)) \wedge\\ \quad allocT(o, OS, \texttt{Incomplete})\ \Rightarrow\\ \quad alive(\widehat{positive}(o, OS), OS)\ \wedge\ \widehat{positive}(o, OS) = OS(o, f) > 0\end{array}$$

Consider the verification of the implementation of method `increment`. For simplicity, let us assume that the **this**-object is the only allocated object. Then, in the prestate, we can assume that the invariant holds for the object, and in the poststate we need to show that `increment`'s postcondition holds and that the invariant holds for **this**. With OS denoting the prestore and OS' the poststore, the corresponding Hoare-triple looks as follows:

$$\begin{array}{l}\{\ \widehat{positive}(this, OS)\ \}\\ \quad \texttt{this.f++}\\ \{\ OS'(this, f) = OS(this, f) + 1\ \wedge\ \widehat{positive}(this, OS')\ \}\end{array}$$

In order to prove that $\widehat{positive}(this, OS')$ holds, we need to use the axiom over `positive` because OS and OS' are not identical, thus the fact that

[2] For simplicity, we ignore the issue of integer overflow in the example.

$\widehat{positive}(this, OS)$ holds is not sufficient. However, the premise of $\mathbf{Ax}_{\widehat{positive}}$ contains an application of $\widehat{positive}$, and in order to utilize the axiom and derive its consequence for term $\widehat{positive}(this, OS')$, we would first need to show that $\widehat{positive}(this, OS')$ holds. And to show that, we would need to use the axiom, which contains symbol $\widehat{positive}$ in its premise...

This means that we ran into an unresolvable cycle, which is due to the appearance of the uninterpreted function symbol in the premise of the axiom.

Note that even declared read and write effects (given in comments) would not help to solve the problem. The write effect of method `increment` overlaps with the read effect of `positive`, thus approaches that make use of effects specifications or dependencies cannot deduce the validity of predicate $\widehat{positive}(this, OS')$ from $\widehat{positive}(this, OS)$.

Similarly, history constraints (also given in comment) cannot solve the problem either. A history constraint is similar to an invariant in that it has to hold in all visible states. However, a history constraint is different in that it contains a predicate that relates two consecutive visible states. The previous visible state is accessible by the use of the `\old` construct.

The specified history constraint does not help because it does not provide enough information to make the use of axiom $\mathbf{Ax}_{\widehat{positive}}$ dispensable. □

As the example demonstrates, if invariants may contain method calls then the proposed axiomatization of pure methods leads to a verification system that is incapable of proving the correctness of methods that modify the store. In order to prevent this, we forbid invariants to contain calls to pure methods.

Remark. Model fields can be seen as parameterless pure methods [100, 22] and, therefore, can be axiomatized the same way as pure methods. Consequently, we also forbid invariants to mention model fields.

Example 6.2. For class `Sequence` in Figure 3.3, the restriction means that the second and third invariants are illegal:

```
invariant isEmpty() ==> length == 0;
invariant !isEmpty() ==> length == rest().length + 1;
```

In order to preserve the properties expressed by these invariants, they have to be reformulated as postconditions. □

6.3.3 Non-recursive Preconditions

In general, our approach allows recursive specifications. However, there is one restriction that we make: preconditions may not be recursive—even if the recursion is well-founded. The two main reasons for this restriction are the following.

First, recursive preconditions would require fix-point computations to define the domains of the corresponding function symbols. This is indicated by Criterion 3, which requires $\langle \mathbf{this}, \mathbf{par}, \mathbf{store} \rangle \in \mathbf{dom}(\mathbf{I}(f))$ to hold under the assumption that $[\mathbf{Pre}(f)]^3_{\mathbf{M}}\theta$ holds. However, if $\mathbf{Pre}(f)$ contained an application of symbol f, then it possibly further constrained $\langle \mathbf{this}, \mathbf{par}, \mathbf{store} \rangle$.

Second, recursive preconditions would lead to a situation similar to invariants that contain method calls. The resulting axioms would contain applications of the symbols that are being defined in their premises.

Checking whether preconditions are recursive or not can be done in a modular way since the declaration of new types leads to the introduction of new function symbols.

We note that requiring preconditions to be non-recursive is not a limitation for practical examples.

6.4 Checking Well-formedness

In this section, we present the proof obligations that are posed on the specification at hand. Our soundness theorem will state that if all proof obligations are valid, then there exist a model for the specification. Consequently, the specification is well-formed.

6.4.1 Incremental Construction of Model

In general, showing the existence of a model requires one to prove the existence of all its functions. To be able to work with first-order logic theorem provers, we approximate this second-order property in first-order logic. We generate proof obligations whose validity in two-valued first-order logic guarantees the existence of a model. However, if we fail to prove them then we do not know whether a model exists or not. That is, the procedure is sound but not complete.

The basic idea of our procedure is to construct a model incrementally. The incremental construction is based on a *dependency graph*.

Definition 6.3. ***(Dependency graph)*** *The nodes of a dependency graph correspond to function symbols and invariants. There is an edge from node n_i to node n_j if and only if the specification of function symbol n_i or invariant n_i applies function n_j.*

In order to capture dynamic method binding, the following rule applies in case of subtyping: If $S \preceq T$ and there is an edge from node n_i to node n_j that corresponds to symbol f_T, and there is a node n_k that corresponds to symbol f_S, then there is also an edge from node n_i to node n_k.

The dependency graph may be cyclic. However, as discussed above, we disallow cycles that are introduced by preconditions.

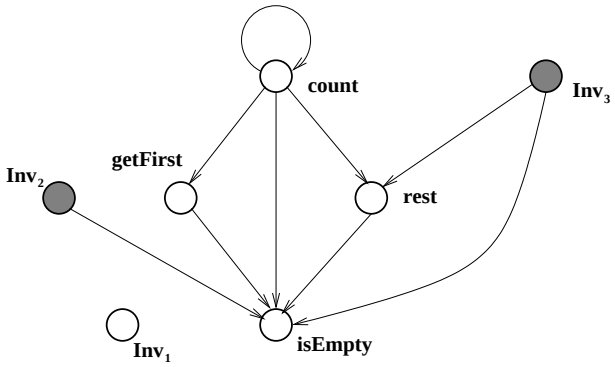

Figure 6.2: Dependency graph for abstract class `Sequence`

Example 6.3. The dependency graph of class `Sequence` introduced in Figure 3.3 is presented in Figure 6.2. The two illegal invariants that contain method calls are marked with darker circles. □

The model is constructed by traversing the dependency graph bottom-up. The starting point is the empty specification $\mathbf{Spec}_0 \triangleq \langle \emptyset, \emptyset, \emptyset \rangle$, for which trivially there is a model $\mathbf{M}_0$. In each step j, a set of nodes $G_j \triangleq \{g_1, g_2, \ldots, g_k\}$ is selected such that if there is an edge from g_i to a node n then either n has already been visited in some previous step (*i.e.*, $n \in G_1 \cup \ldots \cup G_{j-1}$) or $n \in G_j$. Moreover, G_j is chosen such that it has one of the following forms:

1. G_j contains exactly one invariant $\mathbf{Inv}_l \in \mathbf{INV}$.
2. G_j contains exactly one function symbol $f_l \in \mathbf{F}$ and the specification of f_l is not recursive.
3. G_j is a set of function symbols, and the nodes in G_j form a cycle in the dependency graph, that is, they are specified recursively in terms of each other.

Note that in the third form, G_j might contain only one node in case of direct recursion.

The pre- and postconditions and invariants of G_j are called the *specification fragment* of step j, denoted by s_j. In step j, specification $\mathbf{Spec}_{j-1}$ is extended with s_j resulting in $\mathbf{Spec}_j$. To ensure that the model $\mathbf{M}_{j-1}$ for $\mathbf{Spec}_{j-1}$ can be extended to a model $\mathbf{M}_j$ for $\mathbf{Spec}_j$, proof obligations are posed on s_j. Since this construction is inductive, one may assume that all specification fragments processed up to step $j-1$ are well-formed.

It is easy to see that an order in which one can traverse the dependency graph always exists. However, the dependency graph only determines a

partial order between nodes. Thus, there might be multiple specification fragments that could be chosen in a given step. In particular, nodes that correspond to invariants do not have dependencies, because invariants may not contain calls. Therefore, invariants may and should be processed before method specifications, as they might provide information that is useful in later steps.

Note, however, that the well-definedness of an invariant may depend on another invariant. Consider the two invariants:

```
invariant this.o != null;
invariant this.o.value > 0;
```

Well-definedness of the second invariant is only provable if the property expressed by the first invariant can be assumed. Such dependencies between invariants are not traced by the dependency graph. Similarly to other approaches, such as Event-B, we use a user-defined ordering on invariants to resolve this kind of dependency.

At the end of Section 6.4, we will argue that the incremental model construction can be done in a modular way.

6.4.2 Notations for Incremental Construction

Due to the incremental construction of the model, we need to refine the previously introduced notations of specification elements before we could present the proof obligations for the specification fragment of a given step. The refined notations are the following:

• The specification elements processed up to step j are denoted by $\mathbf{Spec}_j$. The three components of $\mathbf{Spec}_j$ are denoted by $\mathbf{Pre}_j$, $\mathbf{Post}_j$, and $\mathbf{INV}_j$. After the last step z of the model construction, we have $\mathbf{Spec}_z \equiv \mathbf{Spec}$.

• $\mathbf{SysInv}_j$ is the conjunction of invariants processed up to step j. Formally:

$$\mathbf{SysInv}_j \triangleq \forall\, o.\ alloc(o, OS) \ \Rightarrow \bigwedge_{Inv\ \in\ \mathbf{INV}_j} Inv$$

After the last step z of the model construction, we have $\mathbf{SysInv}_z \equiv \mathbf{SysInv}$.

• F_j denotes the set of function symbols whose pre- and postconditions have been processed up to step j. After the last step z of the model construction, we have $F_z \equiv \mathbf{F}$.

• We denote the axioms for $\mathbf{Spec}_j$ as follows:

$$\begin{aligned}\mathbf{Ax}^{j}_{f_T} \triangleq\ & \forall\ OS, o, p.\\ & \mathbf{SysInv}_j \wedge \mathit{allocT}(o, OS, T) \wedge \mathit{alive}(p, OS) \wedge \mathbf{Pre}(f_T) \Rightarrow\\ & \mathit{alive}(f_T(o, p, OS), OS) \wedge \mathbf{Post}(f_T)[f_T(o, p, OS)/\mathit{resV}]\end{aligned} \tag{6.2}$$

$$\mathbf{Ax}_{\mathbf{Spec}_j} \triangleq \bigwedge_{f_T \in F_j} \mathbf{Ax}^{j}_{f_T} \ \wedge\ \mathbf{Ax}_{f_{T\preceq}}$$

$\mathbf{Ax}^{j}_{f_T}$ is the definition of the axiom for a method f declared in type T according to specification $\mathbf{Spec}_j$. Note that the axiom $\mathbf{Ax}^{j}_{f_T}$ may be different for different j since $\mathbf{SysInv}_j$ gets gradually strengthened during the construction of the model. Therefore, axiom $\mathbf{Ax}^{j}_{f_T}$ becomes gradually weaker. This is an important observation for the soundness of our approach. In particular, this is the reason why contradicting invariants do not lead to inconsistent axioms, as noted in Section 6.3.1.

After the last step z of the model construction, we have $\mathbf{Ax}^{z}_{f_T} \equiv \mathbf{Ax}_{f_T}$ and $\mathbf{Ax}_{\mathbf{Spec}_z} \equiv \mathbf{Ax}_{\mathbf{Spec}}$.

Well-definedness Operator and Domain Restriction

Although operator *Df* was introduced to generate well-definedness conditions for JML^{--} expressions, the proof obligations that will be posed on the specification fragments use the $\mathcal{L}$ operator. The reason is the shape of the $\mathbf{Inv}_i$ formulas, which encode user-defined invariants. Recall from Section 6.1 that $\mathbf{Inv}_i$ was defined as follows:

$$\mathbf{Inv}_i \ \triangleq\ \mathit{typeof}(o) \preceq T \ \Rightarrow\ \gamma(\mathit{Inv}_i[o/\texttt{this}], OS, \_)$$

Since function *typeof* and operator $\preceq$ are not part of the syntax of JML^{--}, the left-hand side of the implication could not be handled by operator *Df*. Although one could extend JML^{--} and *Df* to include these constructs, we believe it is cleaner to express semantics (in this case, the scope of invariants) with means of the programming logic. Therefore, based on Theorem 5.2 on page 75, operator $\mathcal{L}$ is used to generate well-definedness conditions for specification fragments.

The domain restriction used by $\mathcal{L}$ in step j will be $\mathbf{Pre}_{j-1}$, that is, the domain restrictions of the symbols processed up to step j. This is justified by the fact that in step j, for each $f \in F_{j-1}$ we have:[3]

$$\mathbf{Pre}_{j-1}(f) \ \equiv\ \delta_\gamma(f) \ \equiv\ o \neq \mathit{null} \ \wedge\ \gamma(\mathrm{pre}_f[o/\texttt{this}], OS, \_)$$

[3]Note that the condition in δ_γ contained the additional substitution $p/\texttt{param}$. The substitution is void here as we assume that the explicit parameter of a pure method is p.

Note that invariants are excluded from the domain restrictions, although invariants also restrict the state in which pure methods may be called. The reason for the exclusion is that invariants are assumed to hold in the pre- and poststates, therefore adding invariants to domain restrictions would be superfluous.

Remark. In order to have a uniform handling of symbols, the definition of **Pre** does not contain mappings for the functions of the programming logic (such as *alive* and *loc*) and for the symbols we assumed in the underlying logic (such as *div* and *mod*). However, since **Pre** is used as domain restriction, the mapping of these symbols should be added to **Pre**.

Adding the corresponding mappings is trivial: for functions *loc*, *div*, and *mod* the mappings are presented in Figure 5.3(b) on page 76, all other functions are total and thus mapped to *true*. These mappings can be added to $\mathbf{Pre}_0$ as all domain restrictions are known to be well-defined.

In order to simplify formalizations and examples, in the sequel we omit these mappings from **Pre**.

Since domain restrictions typically do not change, the literature makes them an implicit parameter of $\mathcal{L}$. In our case $\mathbf{Pre}_j$ is used as domain restriction, which may be different in the different steps of the model construction. Therefore, we make domain restrictions an explicit parameter of $\mathcal{L}$ in the sequel.

6.4.3 Proof Obligations

Now we are ready to see the proof obligations that are posed on the three different kinds of specification fragments in step j of the model construction.

6.4.3.1 Invariant $\mathbf{Inv}_l$

In the first case, the specification fragment is some invariant $\mathbf{Inv}_l$. The proof obligation posed on the invariant checks that it is well-defined for all allocated objects in all stores that satisfy the previously processed invariants:

$$\forall\ OS.\ (\mathbf{SysInv}_{j-1} \Rightarrow \mathcal{L}(\forall\ o.\ alloc(o, OS) \Rightarrow \mathbf{Inv}_l, \mathbf{Pre}_{j-1})) \tag{6.3}$$

Example 6.4. We instantiate the proof obligation for a specification fragment from class `Sequence` presented in Figure 3.3. The corresponding dependency graph was presented in Figure 6.2. In the first step of the traversal of the dependency graph, we visit the first (and only admissible) invariant of the class since it has no dependencies. The invariant specifies that `length >= 0`. For this specification fragment, the following proof obligation is posed:

$$\forall\ OS.\ \mathcal{L}(\forall\ o.\ alloc(o, OS) \Rightarrow typeof(o) \preceq \texttt{Sequence} \Rightarrow OS(loc(o, length)) \geq 0, \{\ \})$$

Note that $\mathbf{SysInv}_0$ has been omitted since it is equivalent to *true*. Furthermore, the set of domain restrictions $\mathbf{Pre}_0$ is empty, because no user-defined methods have been processed yet.[4]

After the application of the $\mathcal{L}$ operator and trivial simplifications, we get the following formula:

$$\forall\, OS, o.\ alloc(o, OS) \Rightarrow typeof(o) \preceq \texttt{Sequence} \Rightarrow o \neq \texttt{null}$$

Essentially, we need to prove that the domain restriction of function *loc* is not violated, that is, $o \neq \texttt{null}$. This trivially follows from $alloc(o, OS)$. □

6.4.3.2 Pre- and Postcondition of a Single Function f_l

In the second case, the specification fragment to process is the non-recursive pre- and postcondition of a single function. There are two proof obligations posed on the specification fragment.

Precondition. The first proof obligation checks that the precondition of f_l is well-defined for all allocated receiver objects and alive parameters in all stores that satisfy the processed invariants. Assuming that f_l is declared in type T, the proof obligation looks as follows:

$$\begin{aligned}&\mathbf{Ax}_{\mathbf{Spec}_{j-1}} \Rightarrow\\ &\ \forall\ OS.\ (\mathbf{SysInv}_{j-1} \Rightarrow\\ &\qquad \forall\ o, p.\ allocT(o, OS, T)\ \wedge\ alive(p, OS) \Rightarrow \mathcal{L}(\mathbf{Pre}(f_l), \mathbf{Pre}_{j-1}))\end{aligned} \tag{6.4}$$

Note that $\mathbf{Ax}_{\mathbf{Spec}_{j-1}}$, the axiom system over symbols in F_{j-1}, can be used to discharge the proof obligation. This is necessary because $\mathbf{Pre}(f_l)$ may contain applications of symbols in F_{j-1}.

Example 6.5. In the second step of the traversal of the dependency graph of class `Sequence`, method `isEmpty` is picked because it has no dependencies. The method is processed trivially as it has no specification. Assume the specification of method `rest` is selected as third specification fragment. The precondition of the method is `!isEmpty()`. The corresponding proof obligation is the following:

[4] As noted above, technically $\mathbf{Pre}_0$ contains the mappings for symbols such as *alive* and *loc*. Here, we omit them for simplicity.

$$
\begin{array}{l}
\forall\ OS. \\
\quad (\forall\, o.\, alloc(o, OS) \Rightarrow (typeof(o) \preceq \texttt{Sequence} \Rightarrow OS(loc(o, length)) \geq 0)) \\
\qquad \Rightarrow \\
\quad (\forall\, o.\, allocT(o, OS, \texttt{Sequence}) \Rightarrow \mathcal{L}(\neg isEmpty(o, h), \{\langle isEmpty, o \neq null\rangle\}))
\end{array}
$$

$\mathbf{Ax}_{\mathbf{Spec}_2}$ has been omitted since it is equivalent to *true*. Since method `rest` does not take any parameter, the quantification over p and predicate *alive* is omitted. Note that $\mathbf{Pre}_2$ consists of the mapping for symbol *isEmpty*, which has already been processed.

After the application of the $\mathcal{L}$ operator, one needs to prove that the domain restriction of *isEmpty* is not violated, that is, $o \neq null$. This follows from $allocT(o, OS, \texttt{Sequence})$. □

Postcondition. The second proof obligation checks that the postcondition of f_l is never interpreted as $\perp$ for any result, and that there exists a value which satisfies the postcondition. The property has to be proven for all allocated receiver objects and alive parameters that satisfy the precondition, and in all stores that satisfy the processed invariants:

$$
\begin{array}{l}
\mathbf{Ax}_{\mathbf{Spec}_{j-1}} \Rightarrow \\
\quad \forall\ OS, o, p. \\
\qquad (\mathbf{SysInv}_{j-1} \wedge allocT(o, OS, T)\ \wedge\ alive(p, OS) \wedge\ \mathbf{Pre}(f_l)) \\
\qquad\quad \Rightarrow \\
\qquad (\ (\forall\ resV.\ alive(resV, OS) \Rightarrow \mathcal{L}(\mathbf{Post}(f_l), \mathbf{Pre}_{j-1})) \wedge \\
\qquad\ \ (\exists\ resV.\ alive(resV, OS) \wedge \mathbf{Post}(f_l))\ )
\end{array}
\tag{6.5}
$$

Example 6.6. Postcondition `\result != null` of method `rest` leads to the following proof obligation:

$$
\begin{array}{l}
\forall\ OS, o. \\
\quad (\ (\forall\ o.\ alloc(o, OS) \Rightarrow (typeof(o) \preceq \texttt{Sequence} \Rightarrow OS(loc(o, length)) \geq 0)) \\
\qquad \wedge\ allocT(o, OS, \texttt{Sequence})\ \wedge\ \neg isEmpty(o, OS)\ ) \\
\qquad \Rightarrow \\
\quad (\ (\forall\ resV.\ alive(resV, OS) \Rightarrow \mathcal{L}(resV \neq null, \{\langle isEmpty, o \neq null\rangle\})) \wedge \\
\qquad\ \ (\exists\ resV.\ alive(resV, OS) \wedge resV \neq null)\ )
\end{array}
$$

As before, $\mathbf{Ax}_{\mathbf{Spec}_2}$ is equivalent to *true*. The first conjunct is proved trivially since formula $resV \neq null$ does not contain partial operations. To satisfy the second conjunct, we instantiate $resV$ with o for which we can deduce that $allocT(o, OS, \texttt{Sequence})$ holds, and thus, both $alive(o, OS)$ and $o \neq null$ hold. □

6.4.3.3 Pre- and Postconditions of a Set of Recursively-specified Functions

The third case handles both direct and mutual recursion. The specification fragment to process consists of the pre- and postconditions of a set of functions $G_j \triangleq \{g_1, g_2, \ldots, g_k\}$ with $k \geq 1$.

Besides the properties of the previous case, we need to prove that the recursion in the specification fragment is well-founded. To do so, we use measure functions, assumed to be provided by the user for each recursive function g_i in G_j with signature:

$$\|\cdot\|_{g_i} : \mathbf{Value} \times \mathbf{Value} \times \mathbf{Store} \to \mathbb{N}$$

JML^{--} provides the **`measured_by`** clause to specify the measure function of a method. We require that there is no recursion via measure functions, that is, the definition of measure function $\|\cdot\|_{g_i}$ may only contain function symbols from $G_1 \cup \ldots \cup G_{j-1}$, but not from G_j.

Since preconditions must not be recursively specified (see Section 6.3.3), the proof obligation for the precondition of each g_i is identical to proof obligation (6.4) for the non-recursive case.

In order to prove well-formedness of postconditions, we first need to show that user-specified measures are well-defined and non-negative. For a function g_i with measure clause "**`measured_by`** μ_{g_i}", we introduce a new pure method M_{g_i} with precondition $\mathbf{Pre}(g_i)$ and postcondition $\mu_{g_i} \geq 0$. The dependency graph is extended with a node for M_{g_i} and an edge from g_i to M_{g_i}. Node M_{g_i} is processed like any other node. This allows measures to rely on invariants and to contain calls to pure methods.

Proof obligation (6.6) below for postconditions is similar to proof obligation (6.5), but differs in two ways:

• We have to prove that the recursive specification is well-founded. Due to the extension of the dependency graph mentioned above, we can assume that user-specified measures are well-defined and yield non-negative numbers. Thus, it suffices to show that the measure decreases for each recursive application. We achieve this by using a domain restriction that additionally requires the measure for recursive applications to be lower than the measure *ind* of the function being specified. If the measure *ind* is 0, the domain restriction becomes **false**, which prevents further recursion. Note that the occurrence of *ind* seems to violate the condition that domain restrictions do not contain free variables other than the parameters of the function whose domain they characterize. However, since *ind* is universally quantified, we may consider *ind* to be a constant for each particular application of the domain restriction. One could think of the universal quantification as an unbounded conjunction, where *ind* is a constant in each of the conjuncts.

• For the proof of well-formedness of the specification of a function g_i, we may assume the properties of the functions recursively applied in this specification. This is an induction scheme over the measure *ind*, which is expressed by the assumption in lines 4 and 5 of the proof obligation.

The proof obligation must be discharged for each function symbol $g_i \in G_j$:

$$
\begin{array}{l}
\mathbf{Ax}_{\mathbf{Spec}_{j-1}} \Rightarrow \\
\forall\ ind \in \mathbb{N}, OS, o, p. \\
(\ \mathbf{SysInv}_{j-1} \wedge allocT(o, OS, T) \wedge alive(p, OS) \wedge \mathbf{Pre}(g_i) \wedge \|\langle o, p, OS\rangle\|_{g_i}{=}ind \wedge \\
\quad (\ \bigwedge_{l=1}^{k} \forall\ o', p'.\ allocT(o', OS, T_{g_l}) \wedge alive(p', OS) \wedge \mathbf{Pre}(g_l)[o'/o, p'/p] \wedge \\
\qquad\qquad \|\langle o', p', OS\rangle\|_{g_l}{<}ind \Rightarrow \mathbf{Post}(g_l)[o'/o, p'/p, g_l(o', p', OS)/resV]\ ]\ )\ ) \\
\quad \Rightarrow \\
(\ (\forall\ resV.\ alive(resV, OS) \Rightarrow \\
\quad \mathcal{L}(\mathbf{Post}(g_i), \mathbf{Pre}_{j-1} \cup \{\langle g_l, \mathbf{Pre}(g_l) \wedge \|\langle o, p, OS\rangle\|_{g_l}{<}ind\rangle \mid l \in 1..k\})) \wedge \\
(\exists\ resV.\ alive(resV, OS) \wedge \mathbf{Post}(g_i))\ )
\end{array}
\tag{6.6}
$$

where T_{g_l} is the type in which function g_l was declared in.

Example 6.7. Since proof obligation (6.6) for the postcondition of method `count` (the only recursive specification in the example) is rather large and unreadable, we use a considerably smaller example here, which better highlights how the proof obligation works. We exemplify the proof obligation on the factorial function with the following specification:

```
pure int fact(int p)
   requires p >= 0;
   ensures p == 0 ==> \result == 1;
   ensures p > 0 ==> \result == fact(p-1)*p;
   measured_by p;
```

To simplify the example, we omit the variables for store *OS* and receiver object *o*. Method `fact` is independent of state anyway.

First, we need to prove that measure `p` is well-defined and non-negative. As mentioned above, this is proved by the introduction of a new pure method with specification derived from the precondition and measure of `fact`:

```
pure int M_fact(int p)
   requires p >= 0;
   ensures p >= 0;
```

Proof obligations 6.4 and 6.5 apply to the method, both of which are trivially proven: the conditions are well-defined as they do not contain partial operators, and the precondition implies the postcondition.

Next, we need to show proof obligation (6.6). For brevity, we only show it for the second postcondition, which is the interesting case containing recursion:

$$
\begin{array}{l}
\forall\, ind \in \mathbb{N}, p. \\
\quad (\ p \geq 0\ \wedge\ p = ind \wedge \\
\qquad (\forall\, p'.\ p' \geq 0\ \wedge\ p' < ind\ \Rightarrow \\
\qquad ((p' = 0 \Rightarrow \widehat{fact}(p') = 1\ )\ \wedge\ (p' > 0 \Rightarrow \widehat{fact}(p') = \widehat{fact}(p'-1) * p'))\ )\ ) \\
\qquad\quad \Rightarrow \\
\quad (\ (\forall\, resV.\ alive(resV, OS) \Rightarrow \\
\qquad\quad \mathcal{L}(\ p > 0 \Rightarrow resV = \widehat{fact}(p-1) * p, \{\langle \widehat{fact},\ p \geq 0 \wedge p < ind \rangle\}))\ \wedge \\
\qquad (\exists\, resV.\ alive(resV, OS)\ \wedge\ (p > 0 \Rightarrow resV = \widehat{fact}(p-1) * p))\ )
\end{array}
$$

We need to show that the two quantified conjuncts on the right-hand side of the main implication hold. Proving that the existential holds is straightforward due to the equality and to the fact that values of primitive types are always alive. The other conjunct is more interesting. The only partial operator is $\widehat{fact}$ and after applying the $\mathcal{L}$ operator, the sub-formula simplifies to:

$$\forall\, resV.\ alive(resV, OS) \Rightarrow (p > 0 \Rightarrow p-1 \geq 0\ \wedge\ p-1 < ind)$$

The formula is provable since $p-1 \geq 0$ follows from $p > 0$, and $p-1 < ind$ follows from $p = ind$ in the premise of the proof obligation. □

Soundness

For the presented approach to be sound, the proposed proof obligations have to be shown sufficient to guarantee the well-formedness of specifications. This is stated by our main soundness theorem.

Theorem 6.2. *If a specification* **Spec** *does not contain recursive preconditions and all of the above proof obligations for* **Spec** *hold, then* **Spec** *is well-formed, that is, wf*(**Spec**)*.*

The proof of the theorem runs by induction on the order in which specification fragments are processed, as (partially) prescribed by the dependency graph. For each recursive specification fragment, the proof uses a nested induction on the recursion depth *ind*. The proof is presented in Appendix C.

Modularity

In the realm of object-oriented program verification, modularity is an important aspect of verification techniques. The presented well-formedness

checking and axiomatization technique is modular. That is, adding new types to a program does not invalidate the proofs for the well-formedness criteria of existing methods and invariants. This is due to the following two points.

First, behavioral subtyping ensures that the axiom for an overriding method is stronger than the axiom for the overridden method. Although new types can introduce cycles in the dependency graph that involve existing methods, proofs remain valid since we introduce new function symbols for overriding methods, which thus do not interfere with existing proofs.

Second, the invariants of additional types strengthen **SysInv**, which appears in the premises of proof obligations. Thus, these invariants do not strengthen the proof obligations.

6.5 Related Work

In this section, we discuss related work on the well-definedness checking of specification expressions and the axiomatization of pure methods.

As already seen in Section 5.1, there is a large amount of work in the area of handling partiality in specifications. We listed the three main-stream techniques that are used by different approaches and tools, and summarized the most commonly applied techniques for the elimination of ill-definedness. Since our work applies the elimination technique, we restrict the discussion of related work on well-definedness checking to that technique.

Protective Specifications. Leavens and Wing [78] propose two techniques for the handling of ill-defined specification expressions. One of the techniques is applicable by approaches that use two-valued interpretation and underspecification. The technique poses proof obligations over specification expressions to check if they rely on underspecified values. As noted in Section 5.1.1, relying on such values may be undesired. The other technique is applicable by approaches that use three-valued interpretation. In the sequel, we only look at the latter technique as that is more closely related to our approach.

Leavens and Wing call a procedure specification *partiality-protective* if (1) the precondition is well-defined, and (2) under the assumption that the precondition holds, the postcondition is well-defined. Similar to our approach is that a "definedness predicate" is used with the same semantics as function **wd** (see page 26), and that proof obligations are posed to check if a given specification expression possibly evaluates to $\bot$. In fact, the proof obligations are similar to those parts of formulas (6.4) and (6.5) that are concerned with well-definedness.

The main difference is that the approach of Leavens and Wing is presented in a two-tiered setting, hence, dependencies between procedures need not be handled.

Specification languages often provide means to protect specification expressions from being ill-defined or underspecified. For instance, traits of the Larch Shared Language introduce predicates such as `isValid` and `between`, the OCL Standard Library contains a built-in operator `isDefined`, and the specification language of Caduceus has a predefined predicate `\valid_range`. Furthermore, the use of conditional operators (such as `&&` in Java or **`and then`** in Eiffel) also facilitate the protection of specification expressions.

Theorem Provers

Well-Definedness. As mentioned earlier, PVS and CVC Lite are two theorem provers that generate well-definedness conditions to eliminate ill-defined terms and formulas. In PVS, well-definedness checking is combined with type checking. A partial function is modeled as a total function whose domain is a predicate subtype [117]. This makes the type system undecidable requiring Type Correctness Conditions to be proven. PVS uses the $\mathcal{L}$ operator for the generation of these conditions because $\mathcal{D}$ was found to be inefficient [117].

Recall that the use of the $\mathcal{L}$ operator has the consequence that the interpretation of commutative logical operators become non-commutative. Thus, well-defined formulas may get rejected. Although the manual re-ordering of sub-formulas can resolve such cases, Cheng and Jones [30], and Rushby *et al.* [117] give examples for which even manual re-ordering does not help, and well-defined formulas are inevitably rejected by $\mathcal{L}$.

CVC Lite [12] uses the $\mathcal{D}$ operator for the well-definedness checking of formulas. This decision eliminates the above mentioned problem, furthermore, Berezin *et al.* [17] reckon that if formulas are represented as DAGs, then the worst-case size of $\mathcal{D}(\phi)$ as a DAG is linear with respect to the size of ϕ. Therefore, the exponential blow up does not cause a problem for CVC Lite.

Schieder and Broy [120] propose a different approach to the checking of well-definedness of formulas. They define a formula under a three-valued interpretation to be well-defined if and only if its interpretation yields **true** both if $\bot$ is interpreted as **true**, and if $\bot$ is interpreted as **false**. Although checking well-definedness of formulas becomes relatively simple, the interpretation may be unintuitive for users. For example, formula $\bot \vee \neg\bot$ is considered to be well-defined.

In theorem provers, dependencies typically have to be resolved by the user. For instance, a definition may only refer to a function symbol if it has already been declared and defined before. Therefore, it is rather straightforward what pieces of information may be used to prove well-definedness conditions.

Consistency. Theorem provers typically ensure consistency of theories by restricting definitions to form conservative extensions. On the other hand, user-declared axioms are typically not checked, which are therefore a known source of inconsistency. Moreover, theorem provers typically require users to resolve dependencies themselves by ordering the elements of a theory appropriately. Recursion (both direct and mutual) is usually supported, and well-foundedness is enforced by posing proof obligations that are based on user-defined measure functions.

Theorem provers that ensure consistency this way include LCF and related systems such as HOL, Isabelle, and ACL2.

Our approach differs in that specifications are not restricted to conservative extensions, which would not fit for program specifications. Instead, we allow arbitrary specifications and reject those for which the existence of a model cannot be proven. Furthermore, dependencies are resolved by the automatic construction of dependency graphs, which lessens the efforts of writing specifications.

Design Languages. Different design languages apply different approaches to the handling of partial functions. For instance, Z and Larch uses underspecification, VDM uses a three-valued logic, and Event-B generates proof obligations to eliminate ill-defined terms.

Design languages typically do not restrict specifications to conservative extensions. Instead, similarly to our approach, consistency is guaranteed by posing proof obligations over specifications.

In his introductory book on Z [129], Spivey does not discuss the issue of Z specifications being possibly infeasible. However, infeasible specifications may cause inconsistent reasoning in Z [132]. Hall *et al.* [56] present a method for showing the existence of a model for Z specifications. The idea is similar to ours in that proof obligations are posed that require the satisfiability of specifications. The case of recursive specifications are not discussed. Valentine [132] proposes a similar idea, and notes that the existential quantification can be removed in non-recursive cases when result values of functions are defined by equality. In such cases, the expression on the right-hand side of the equality is a witness for satisfiability. Valentine discusses recursive specifications but does not give a systematic technique for their handling.

VDM also poses proof obligations on specifications to ensure their satisfiability [68]. However, it is not clear whether and how dependencies between specification elements influence satisfiability checking. Furthermore, no proof obligation is presented to ensure well-foundedness of recursive specifications. However, this might be a consequence of VDM using a three-valued logic, and thus not having to eliminate ill-defined function specifications. VDM also allows the use of unchecked and thus possibly inconsistent axioms.

Event-B requires that every user-defined specification element (*e.g.*, invariant, guard, action, and variant) is well-defined. Tools that implement the methodology of Event-B use the $\mathcal{L}$ operator because the $\mathcal{D}$ operator proved to be inefficient in practice [16, 4]. In Event-B, the well-definedness of an invariant may depend on other invariants. Such dependencies have to be resolved by their manual re-ordering. The well-definedness of other parts of a specification may also depend on invariants, but not the other way around. Therefore, the well-definedness of invariants can be checked first, and if indeed found well-defined, assumed in latter steps. The well-definedness of the specification of an event does not depend on other events, thus dependencies between events need not be considered when checking their well-definedness.

In Event-B, satisfiability of specifications has to be proven. That is, in contrast to our approach, the satisfiability of invariants has to be proven, too. Proof obligations ensure that initialization events satisfy the invariants and that other events preserve them. Although the actions of events consist of assignments, they can be seen as postconditions. In fact, generated proof obligations reflect actions as so-called before-after predicates, which correspond to postconditions that relate values using old-expressions. The satisfiability of before-after predicates is ensured by requiring witnesses to be exhibited. For an event that is supposed to converge, it has to be proven that the event decreases a user-defined variant and that the variant is a natural number.

The checks on before-after predicates and variants are similar to the checks that our approach performs on postconditions. The main difference is that the language of Event-B does not allow dependencies between events, guards, and actions. Hence, there is no need for an incremental checking and for a complicated proof obligation for convergent events.

While invariants and event specifications are checked, users can add arbitrary axioms whose consistency is not.

LSL, the language-independent algebraic specification language of the Larch family, is checked for certain semantic properties, including consistency. However, the check is not sound and is mainly used for automatically finding inconsistencies [52]. Running the check for some time without being warned of an inconsistency increases one's confidence in the correctness of a library.

As mentioned earlier, OCL has a three-valued interpretation. HOL–OCL [24, 23], a proof environment for the analysis of UML/OCL specifications, embeds OCL into Isabelle/HOL [106]. HOL–OCL does not eliminate ill-defined specifications but preserves OCL's three-valued interpretation and defines a "weak definedness" operator (similar to $\mathcal{D}$), which is built into the calculi of the proof environment.

To consider a UML/OCL specification to be consistent, HOL–OCL poses three consistency requirements on OCL specifications [24]. The requirement

on postconditions expresses the same property that proof obligation (6.5) enforces. The other two proof obligations ensure that invariants are satisfiable and that preconditions are satisfiable under the assumption that invariants hold. Although both requirements can catch bogus specifications, our approach does not enforce these two properties. As discussed above, infeasible invariants make generated axioms void and thus do not jeopardize soundness of our approach. The same holds for infeasible preconditions. There are no special checks defined for recursive specifications, thus it is not clear whether and how the three requirements ensure well-foundedness of such specifications.

Program Verifiers. ESC/Java did not check well-definedness of specification expressions, and since it did not allow pure-method calls in specifications, there was no need to axiomatize method specifications.

Well-definedness checks were added to ESC/Java2 as described by Chalin in [28]. The proposed well-definedness operator is based on the $\mathcal{L}$ operator for logical connectives. Although ESC/Java2 allows specifications to contain calls to pure methods, Chalin does not describe how dependencies among specification elements are handled. Thus, it is not clear whether his approach is capable of discharging conditions that contain uninterpreted function symbols.

ESC/Java2 axiomatizes method specifications according to the formalization of Cok [32]. Consistency of the axiom system is not ensured, which can lead to unsound reasoning. For instance, a program that contains one of the two methods `wrong` or `direct` from Example 3.6 on page 37 turns the axiom system of ESC/Java2 inconsistent.

As mentioned in Chapter 4, Krakatoa and Caduceus do not allow the use of pure methods. The function and predicate symbols that one can introduce on the source level are considered to be total. The same applies for operators of the programming language, such as field and array accesses. The technique of underspecification is used on "improper" applications of these symbols an operators.

The consistency of axioms that one can introduce are not guaranteed in Caduceus and Krakatoa [90]. Marché lists four different ideas to overcome this problem. The idea closest to our solution is to extract Coq templates out of stated axioms, and then manually fill in the templates by providing concrete definitions for the newly introduced function and predicate symbols. Thereby one would give an actual model for the axiom system, which guarantees consistency. The other three ideas also aim to find models, in particular, by the construction of new inductive data types or by the re-use of pre-defined data types. These four ideas are not worked out in detail and are considered future work [90].

The KeY System applies the technique of underspecification and there-

fore does not check well-definedness of specifications. KeY does not generate axioms to encode properties of pure methods. Instead, its calculus contains a rule that replaces an invocation to some pure method m by its postcondition provided that the precondition of m holds in the state of the invocation. This solution prevents problems with ill-founded specifications. For instance, method `direct` in Example 3.6 on page 37 does not lead to unsound reasoning, instead, the rule that replaces invocations to `direct` becomes applicable infinitely many times. On the other hand, KeY does not check feasibility of method specifications before enabling the rule to replace invocations. Therefore, the KeY System becomes unsound if method `wrong` from Example 3.6 is used in specifications.

Jack also applies the technique of underspecification. The tool axiomatizes pre- and postconditions of pure methods separately. This separation ensures that axioms are only instantiated when a pure-method call occurs in a given verification condition—as opposed to being available to the theorem prover at any time. However, since Jack does not check consistency, unsoundness can still occur by the use of axioms. Jack does not support recursive specifications.

At the time of writing, Spec# only checks the well-definedness of invariants. The applied well-definedness operator is similar to *Df* [87]. For the checks, the axioms that are generated over pure methods may be used. Since the well-definedness of pure-method specifications is not checked, such axioms encode potentially ill-defined expressions. However, this does not jeopardize soundness as such expressions will only be underspecified. For instance, consider a method `m` with an integer parameter `p` and a specification that requires `p` to be non-negative and specifies the return value to be 10. Spec# does not filter out ill-defined calls to `m` in specification expressions like `m(-1)`. However, one will not be able to deduce anything about the result of the call, because the premise of the extracted axiom is false, making the axiom void.

Since Spec# uses fully automated theorem provers, it does not generate proof obligations with existential quantifiers to ensure the existence of possible result values for method specifications. Instead, based on a heuristics that inspects the shape of postconditions, it tries to guess proper values and attempts to prove that at least one of the guessed values indeed satisfy the method specification at hand [82].

Spec# ensures well-foundedness of specifications by defining an ordering and proving that every call in a method specification is ordered below the method being specified [82]. The ordering has two components. First, the topology of object structures is taken into account based on the acyclic ownership relation. If the specification of some method contains a call with receiver object o, then the call is ordered below the method being specified if o has more owners than the **`this`**-object. That is, o is "deeper" in the ownership structure, which, due to the acyclicity of the ownership relation

ensures well-foundedness. The second component of the ordering is based on user-defined measures, similar to our approach.

Spec# does not check the consistency of method specifications incrementally, thus, no knowledge about the properties of pure methods can be used during the checking process. This is a source of incompleteness.

Jacobs and Piessens [65] do not address the issue of well-definedness of expressions that are used in the axiomatization of inspector methods. The consistency of axioms generated over inspector methods is ensured by simple syntactic means. Satisfiability is guaranteed by requiring that method bodies of inspector methods are of the form **`return`** E. The axiomatization technique then extracts expression E to specify the value of the corresponding function symbol. Recursive specifications are forbidden.

Dynamic Frames. In VeriCool [126], the well-definedness of method specifications is checked. For each method, a proof obligation is stated that requires the precondition to be well-defined, and the postcondition to be well-defined provided that the precondition holds. These are similar to the requirements our approach poses too (cf. Criteria 2 and 4 in Section 6.2). However, Smans *et al.* do not address the issue of dependencies: it is not clear exactly what information (such as invariants and axioms over other methods) can be used to discharge the well-definedness conditions. Furthermore, recursive specifications are forbidden.

VeriCool uses the same idea to ensure satisfiability of dynamic frames as the inspector-methods approach.

Recall that Dafny [81] does not support pure methods, instead, functions are introduced for the purpose of abstraction. However, the issues raised by their well-definedness checking are the same as for pure methods. The well-definedness operator of Dafny is defined the same way as Df for all constructs that are common in the expression syntax of JML^{--} and Dafny. There are two main differences between the way Dafny and we check the well-definedness of specifications.

First, in Dafny, the well-definedness of the specification of some function m has to be proven whenever the axiom generated over symbol $\widehat{m}$ is to be used. In contrast, our approach requires one to prove the well-definedness of m's specification once and for all. Consequently, our approach requires specifications to be well-defined in all possible states, while Dafny requires specifications to be well-defined only in states in which they are actually used for reasoning. For instance, while our approach rejects expression `a/b` if we cannot deduce that `b` is non-zero, Dafny allows one to use the enclosing specification as long as it is used in states where `b` is non-zero. This reflects different design decisions: Dafny does not try to catch possibly ill-defined specifications as we do, instead, uses specifications for reasoning as long as they are well-defined.

Second, in contrast to our approach, the well-definedness checking of

Dafny does not take dependencies into account. Hence, the well-definedness of specifications that contain function applications cannot be proven.

In Dafny, the functions are defined by strongly-pure expressions. These expressions also serve as witnesses for the satisfiability of the functions. Well-foundedness of the expressions is ensured by defining the set of locations that the function possibly reads as the measure. Since Dafny does not take dependencies into account, proof obligations that ensure well-foundedness have to be proved without knowledge about the properties of the functions.

Smans *et al.* [128] do not address the issue of well-definedness of expressions that are used in the axiomatization of dynamic frames. This situation is improved for implicit dynamic frames [127] by posing proof obligations similar to ours on the specifications of pure methods and dynamic frames.

In both approaches, satisfiability of dynamic frames is ensured in a similar way as in the inspector-methods approach. Recursive specifications are forbidden in the approach presented in [128]. The approach of implicit dynamic frames admits recursive specifications, and to show well-foundedness, the *required access set* of dynamic frames is used as measure [127]. The required access set of a dynamic frame is an upper bound on the set of locations possibly accessed by a frame.

Ballet does not attempt to ensure the well-definedness of Eiffel contracts. This is because Eiffel follows the approach of "blame assignment" in this respect, too. For instance, the caller of a feature has to make sure that the precondition of the callee is well-defined [122]. It is not clear how Ballet handles situations where this assumption is violated.

Ballet does not address the issues of consistent axiomatization of specifications [122, p.164].

Model Fields. The value of a model field is not looked up in the store, but is determined by the **`represents`** clause of the field, which specifies its possible values by a relation to other locations. A bogus specification might be unsatisfiable for any value, thus an improper encoding of model fields may lead to inconsistency.

The work of Leino and Nelson [79, 86], and that of Müller [100] do not tackle the issue of well-definedness and inconsistency. Therefore, bogus specifications may result in an inconsistent axiom system.

Breunesse and Poll [22] address the consistency problem for model fields. They propose two solutions. Their first solution uses existential quantification to ensure that the representation relation of a model field is satisfiable. However, the resulting encoding yields **false** for every JML expression E that contains a model field whose representation cannot be satisfied, even if E is a tautology. This may not be desired. The second solution transforms model fields into pure methods. This solution requires a sound encoding of pure methods, which is not addressed. Breunesse and Poll do not consider recursive representation relations.

Leino and Müller [84] generate axioms over model fields by extracting specified representation relations. Consistency of the axioms is ensured by requiring the existence of a value that satisfies the specified relations. Well-foundedness of **`represents`** clauses is guaranteed by allowing recursive accesses to model fields only along the acyclic ownership relation. Well-definedness is not checked.

Chapter 7

Implementation

We have implemented the techniques presented in the previous chapters in order to evaluate them [33, 133]. Our implementation extends the Spec# language and verification system. This decision was mainly due to (1) the features of C# that provide means to "mimic" language extensions in a simple way; (2) Spec#'s rich infrastructure that one can easily get hold of (*e.g.*, interfacing different back-end provers); and (3) the amount of experience we already had had with Spec#. However, in general, we could have chosen other verification tools as well, for instance, ESC/Java2 or Krakatoa.

The Spec# System in a Nutshell

The Spec# system consists of three main components [10, 8]: the Spec# language, the Spec# compiler, and the static verification component, Boogie. The pipeline of the tool (containing our extensions) is sketched in Figure 7.1.

The Spec# language is an extension of C#. The most important extensions are method specifications, invariants, a non-null type system, and an ownership model. Just as C# programs do, Spec# programs compile to CIL bytecode, the bytecode format of the .NET Platform.

The compiler analyzes the Spec# source code and generates bytecode that (1) contains inlined code for method specifications and invariants; and (2) preserves all specifications, including non-nullness and ownership information. The inlined code is used for runtime assertion checking, while the preserved specification is used by Boogie.

Boogie translates the bytecode into an intermediate language, BoogiePL. BoogiePL [37] is a simple imperative language with procedures whose implementations consist of a few kinds of statements. BoogiePL code may also contain function declarations and axioms. In fact, Boogie encodes pure methods by uninterpreted function symbols and axioms over these symbols on the level of BoogiePL. A BoogiePL program goes through different translation steps to yield verification conditions in first-order logic, which are fed to a fully automated theorem prover.

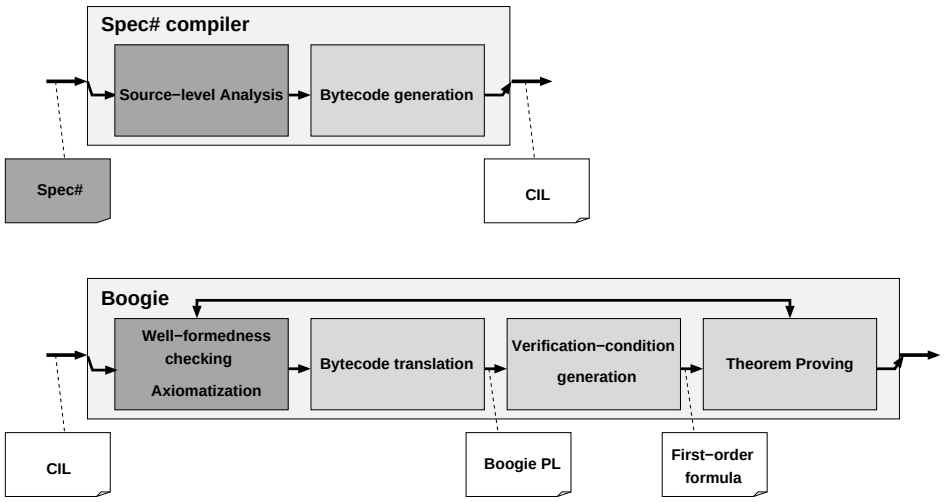

Figure 7.1: The extended pipeline of the Spec# system

Extensions to the System

Our implementation extends all three main components of the Spec# system. The three parts of the system that our implementation modified or added are drawn in dark gray in Figure 7.1.

Language. For the extension of the Spec# language we used attributes. In C#, attributes allow one to associate some data with program elements such as classes, methods, or fields. Therefore, attributes provide a simple means to mimic new modifiers and specification clauses. The effort of adding a new attribute is significantly smaller than adding a new modifier or clause, because the grammar of the language remains the same.

Our implementation extends the Spec# language with the following four attributes:

- **`ResultNotNewlyAllocated`** and **`NoReferenceComparison`** directly correspond to the modifiers introduced in Section 4.4.1 for eliminating the problems with reference comparison.

- attributes **`Measure`** and **`MeasureRep`** are introduced to specify measure functions for recursive specifications. The **`Measure`** attribute corresponds to the **`measured_by`** clause discussed in Section 6.4.3. The argument of the attribute must be of type **`int`**. Attribute **`MeasureRep`** specifies that the measure of recursive calls is based on the acyclic ownership relation, and the receiver of recursive calls must be referenced through a *rep*-field.

Compiler. The compiler was modified in two ways. First, the four attributes were added and the information on user-defined measures were made to get preserved in generated bytecodes. Second, the simple static checks described in Section 4.4.1 were added. A warning message is reported if the checks fail.

Verifier. Boogie uses one store for the encoding of pure methods. This corresponds to our simplified encoding presented in Section 4.4. Therefore, the encoding of Boogie was left unchanged.

Boogie has been modified such that the technique described in Chapter 6 is applied before bytecode would get translated to BoogiePL. That is, we build the dependency graph of the program at hand, and traverse it to check the well-formedness of invariants and method specifications.

Proof obligations are directly sent to the theorem prover, and if a proof obligation fails then an error message gets reported. Failed proof obligations can be queried for further analysis. If all proof obligations are discharged for the specification of some method, then the corresponding axiom (see formula (6.2) on page 90) is added to the theorem prover.

After a successful traversal of the dependency graph, axioms of the form of (6.1) on page 82 are passed on to the bytecode translator. The translator turns them into axioms in BoogiePL, thereby making them available for the verification of the source code.

All checks performed by the compiler and the verifier can be turned off by a command-line option.

Remark. Our implementation does not adapt the axioms generated over pure methods to the invariant semantics of Spec# [83]. This is so because the main purpose of our implementation was to evaluate the technique described in Chapter 6. An adaptation to Spec#'s invariant semantics is also possible.

Adaptation for Automatic Theorem Provers

The proof obligations presented in the previous chapter are sufficient to show the well-formedness of a specification. However, they are not well-suited for automatic theorem provers for two reasons. First, the proof obligations that ensure consistency of postconditions (proof obligations (6.5) and (6.6)) contain existential quantifiers, for which automatic theorem provers often do not find suitable instantiations. Second, the proof obligations that ensure well-foundedness of recursive specifications (proof obligation (6.6)) is in general proved by induction over *ind*, but induction is not supported well by automatic theorem provers.

Our implementation adapts these proof obligations the following ways to make well-formedness checking practical with automatic theorem provers.

Consistency. Similar to the heuristics of Spec# proposed by Leino and Middelkoop in [82], we inspect postconditions of pure methods to identify *witness expressions*. A witness expression is an expression that satisfies the postcondition of the pure method at hand. In contrast to the approach of Spec#, a witness expression surely satisfies the corresponding postcondition, thus there is no need to verify that. This also means that our checks are more conservative.

Our implementation considers an expression E to be a witness expression if (1) the method has exactly one **`ensures`** clause and it has the form "`\result` $\odot$ E" or "$E \odot$ `\result`", where $\odot$ is one of the reflexive operators `==`, `>=`, `<=`, `==>`, or `<==>`, and (2) E is an expression that does not contain the literal `\result` [33].

In order to admit more complex specifications, the syntactic checks are extended to allow multiple postconditions of the form "**`ensures`** P_i `==>` E_i", provided that: (1) in no state do two premises hold simultaneously and (2) every consequence contains a witness expression. The first property is enforced by proof obligations. This extension makes, for instance, the postcondition of method `count` admissible in Figure 3.3 on page 36.

By automatically discovering witness expressions, the existential quantifier can be eliminated from proof obligations (6.5) and (6.6).

Well-foundedness. Proof obligation (6.6) in general requires induction. However, induction is only needed if a pure method is specified recursively *and* the recursive call occurs as an argument to a partial operation, such as division or a method call. Therefore, we syntactically forbid recursive calls to occur as arguments of partial operations.

Furthermore, our implementation forbids mutual recursion, which can be easily detected by the inspection of the dependency graph. This restriction further simplifies the form of the proof obligation: lines 4 and 5 can be omitted because one function can be processed at a time. Although forbidding mutual recursion is conceptually a major restriction, in practice, the use of mutual recursion (that is both useful and well-founded) is not common.

Our implementation automatically discovers and admits recursion over the acyclic ownership relation: if the receiver of a recursive call is directly owned by the **`this`**-object, then the recursion is considered to be well-founded [33]. This further simplifies proof obligation (6.6) as the decrease of the measure does not have to be proven.

Evaluation

Novelty. To the best of our knowledge, our tool was the first one that performed well-formedness checks on invariants and method specifications for an object-oriented specification language. In particular, the incremental way of checking specification constructs was novel in the realm of program verifiers.

In the meantime, the checking mechanism of the implicit dynamic frames approach by Smans *et al.* [127] has been adapted to perform an incremental model construction, too. Since our technique is not bound to a particular logic or theorem prover, other program verifiers could adapt it as well.

Implementation. As Figure 7.1 indicates, our technique is mostly independent of the rest of Boogie's pipeline. Therefore, our implementation only modifies existing Spec# code in a few points, and mostly consists of newly introduced classes. This makes our well-formedness checking easily "plugable" into new releases of the Spec# system.

Annotation Overhead. Besides measure specifications for recursive specifications, users manually have to add attributes `ResultNotNewlyAllocated` and `NoReferenceComparison`. We believe that more accurate analyses that go beyond our syntactic checks could lessen this burden. For instance, a points-to analysis could help to infer attribute `ResultNotNewlyAllocated`. Still, based on our experience, the annotation overhead seems to be acceptable.

After the implementation of the admissibility checks, we used the source code of Boogie itself as one of the benchmarks [33]. At that time in 2006, Boogie comprised some 45K lines of Spec# code. Although the specification of the Boogie source code was far from being complete and expressed mainly simple properties, all uses of pure methods in the source code passed the admissibility checks. We only had to mark two methods with `ResultNotNewlyAllocated` (one in Boogie and one in mscorlib) and none with `NoReferenceComparison`. Furthermore, we did not encounter specifications that were rejected because of the over-approximative nature of the analysis presented in Section 4.4.1.

Proving Capabilities. As discussed in the related work section of the previous chapter, there are only a few tools that check the well-formedness of specifications. Among those few tools, in our opinion, Spec# is the one that provides the best support for the checks. Therefore, we briefly compare our implementation to the current version of Spec#.[1]

As mentioned before, Spec# checks well-definedness only for invariants, and the checks may assume the axioms that are generated over pure methods. Consequently, in contrast to our approach, Spec# does not need to use a dependency graph for well-definedness checking. On the other hand, axioms generated over pure methods may contain expressions that are ill-defined. However, such axioms do not lead to inconsistency.

The main difference between Spec#'s and our approach to the check-

[1] At the time of writing, the latest public release was v1.0.21125 from November 2008.

ing of well-definedness is that our approach warns users about possibly ill-defined specifications before the actual proving process would begin, while Spec# encodes such specifications and considers them underspecified when used for reasoning. We have decided for our design because we think that "debugging" specifications as early as possible is beneficial for users.

As mentioned above, Spec# uses heuristics to guess potential witnesses for method specifications [82]. These heuristics are more powerful than our approach to consistency checking, described above.

Recall that our implementation considers a method specification consistent either if it has exactly one ensures clause that has a witness or if it has multiple conditional ensures clauses where the conditions are disjoint and each clause has a witness.

The heuristics of Spec# are not limited to conditional ensures clauses but are defined over all logical connectives, thereby providing a more flexible solution. As future work, we plan to adapt our implementation to use the heuristics of Leino and Middelkoop.

Spec# currently does not implement the measure clause described in [82]. Therefore, recursion in specifications is only allowed along the acyclic ownership relation. This means that our implementation is more powerful regarding recursive specifications.

It is important to note that in the presence of measure clauses, most probably Spec# would also need to track dependencies among pure methods: (1) In order to prove that specified measures are non-negative and decrease by every recursive call, the axioms over pure methods might be needed. The lack of these axioms would be a source of incompleteness. (2) However, because of the possibility of cyclic reasoning, it would be unsound to use the axioms before having checked that the specifications from which the axioms were extracted are well-founded.

Experience. Although our implementation works well for smaller programs, at the time of writing, it has not yet been tested by larger examples or case studies. It remains future work to develop challenging examples to see the usability and practicality of our implementation.

Part II

Faithful Mapping of Model Classes

Chapter 8

Motivation

In Part I, we have seen how the pure methods of a program can be used to write abstract specifications. For instance, in Figure 1.1 on page 7, method `isPremium` of class `Account` was used to specify method `addBonus` of the class. Another common way of writing abstract specifications is to specify implementations in terms of well-known mathematical structures, such as sets and relations. This technique is applied, for instance, in VDM, Larch, and OCL. While these approaches describe the mathematical structures in a language that is different from the underlying programming language, the one-tiered JML simplifies the development of specifications by describing the structures in an object-oriented manner through *model classes* [31].

A model class is immutable and contains only pure methods, which provide an interface to the mathematical structure that the class represents. While model classes are useful for specification purposes, they pose the same problem for verification as the pure methods of programs: verifiers have to encode model classes in the underlying theorem prover.

Although the technique described in Part I could be applied for the encoding of model classes, it would not be optimal, mainly for the following three reasons. (1) The tactics of a theorem prover are optimized for theories that are part of the prover's theory-library, and not for the axiomatization of model-class specifications. (2) As seen in Part I, it is difficult to ensure consistency of such encodings, in particular, in the presence of recursive specifications. (3) The encoding technique can ensure consistency of specifications, but not their semantical correctness. While we cannot do better for pure methods that are written for a given domain (*e.g.*, method `isPremium` in class `Account`), the situation is different for model classes. For instance, we have a good understanding about the semantics of method `union` in the model class that represents mathematical sets.

Therefore, previous work [29, 75, 76] proposes to map model classes and their pure methods directly to theories of the underlying theorem prover. This is possible because model classes are in fact very similar to mathemat-

```
class SingletonSet {
    Object value;
    model JMLObjectSet _set;
    represents _set <- new JMLObjectSet(value);

    void setValue(nullable Object o)
      ensures _set.has(o);
      assignable value;
    { value = o; }

    // other constructors and methods omitted
}
```

Figure 8.1: Specifying `SingletonSet` using model class `JMLObjectSet`

ical structures. Their objects are immutable, their operations are side-effect free, and equality is based on their state rather than object identity. Therefore, instances of model classes behave like mathematical values rather than heap-allocated objects. This view greatly simplifies reasoning about model classes.

Example 8.1. Figure 8.1 shows the use of model class `JMLObjectSet` for the specification of class `SingletonSet`. Model class `JMLObjectSet` is a prefabricated class that encodes a mathematical set of objects through its pure methods. In order to use the model class in the specification of class `SingletonSet`, model field `_set` is declared. The field represents the abstraction of an instance of type `SingletonSet` as specified by the **`represents`** clause:[1] a singleton set containing the object referenced by field `value`. Method `setValue` is specified in terms of the model field and `JMLObjectSet`'s pure method `has`, which checks for set membership.

Let us see how one would prove the postcondition of method `setValue` in store OS. Due to our semantical understanding of method `has`, expression `_set.has(o)` can be mapped to $o \in OS(\texttt{this}.\_set)$. To determine the value of location `this._set` in store OS, we map the **`represent`** clause of field `_set` to the singleton set, yielding term $\{OS(\texttt{this.value})\}$. Given the assignment in the body of the method, we obtain the following proof obligation for `setValue`: $o \in \{o\}$, which is trivial to prove.

The proof obligation above is considerably better suited for verification than the proof obligation $\widehat{has}(\widehat{JMLObjectSet}(o, OS), o, OS)$ that one would obtain by encoding the expression with function γ, introduced in Chapter 4. □

[1] Note that the syntax of JML^{--} does not contain the **`represents`** clause. It is used here for demonstration purposes only. The same holds for the **`assignable`** clause of method `setValue`.

Previous work discusses only the mapping of method signatures, but ignores their contracts. With this approach, the meaning of `has` is given by the definition of symbol '∈' of the underlying theorem prover, and not by the contract of `has`. This is problematic if there is a mismatch between the contract and the semantics of the operation given by the theorem prover. Static program verifiers might produce results that come unexpected for programmers who rely on the model class contract. The results may also vary between different theorem provers, which define certain operations slightly differently (for instance, division by zero yields 0 in Isabelle, while it is not admissible in PVS). Moreover, the result of runtime assertion checking might differ from that of static verification if the model class implementation used by the runtime assertion checker is based on the model class contract.

Example 8.2. To illustrate the possible discrepancies between static verification and runtime assertion checking, suppose that we had the following hypothetical specification for one of the constructors of `JMLObjectSet`:

```
JMLObjectSet(nullable Object e)
   ensures (e == null ==> \result.isEmpty()) &&
           (e != null ==> \result.has(e));
```

That is, the constructor returns an empty set if the argument is `null`, and a set that contains `e` otherwise.

With this behavior of the constructor, the runtime assertion checker would report an error for the call `mySingletonSet.setValue(null)` because in the poststate of `setValue`, the set `_set` would be empty and, thus, would not actually contain `null`. That is, `_set.has(null)` yields false. However, as we have argued above, the proof obligation obtained by applying the mapping can be trivially proven. This discrepancy would be due to the mapping of the one-argument constructor to the term {e}, which would alter the semantics of the specification in the case when `e` is `null`. □

8.1 Contributions

In Part II of the thesis, we show how model classes can be mapped to theorem provers without semantic mismatches. The main contribution of our work is a technique for proving that the mapping of a model class to a mathematical structure defined by the theorem prover is *faithful*, that is, the model class and the structure indeed correspond to each other in their properties.

To prove faithfulness of a mapping, we apply in two steps the technique of *theory interpretation*, introduced in Section 2.3.

In the first step, we attempt to formally prove that the specified mapping of a model class defines a standard interpretation of the theory formed by the specification of the model class in the theory of the corresponding

structure. The proof is based on showing that the three sufficient obligations presented on page 27 hold. If the proof attempt succeeds then, according to Theorem 2.2 on page 28, *consistency* of the model-class specification is guaranteed. Consistency is only relative to the consistency of the target theory. However, theory-libraries of theorem provers are unlikely to contain inconsistencies.

In the second step, we attempt to "reverse" the specified mapping and attempt to prove that the resulting mapping defines a standard interpretation of the target theory in the specification of the model class. If the proof attempt succeeds then, according to the definition of standard interpretation, *completeness* of the model-class specification is guaranteed. Again, completeness is only relative to the corresponding theory. However, theories of theorem provers typically define a rich set of properties.

We show that the backwards mapping that is used in the second step must not be an arbitrary one, but has to be one that is indeed the reverse of the specified mapping. We define a condition that ensures the existence of such a backwards mapping.

In the sequel, we will refer to these two steps of the faithfulness proof as the *consistency proof* and the *completeness proof*.

In practice, often consistency and completeness cannot be proven because the related model class and structure does not perfectly match and no mapping can be given for a model-class method or a symbol of the structure. In Chapter 10, we introduce the notion of *observational faithfulness*, which admits a relaxed version of the completeness proof in that it does not require all symbols of a structure to have a corresponding functionality in the model class. As discussed in Chapter 10, observational faithfulness is a sufficient result for the sound use of `mapped_to` clauses.

Once (observational) faithfulness of the mapping of a model class has been proven, the mapping can be used for program verification without worrying about semantic discrepancies. In particular, the hypothetical discrepancy between the static verification of `setValue` and runtime assertion checking shown in Example 8.2 would be detected during the consistency proof for `JMLObjectSet`.

Our approach leads to important results beyond semantical correspondence and simplified reasoning. Model class contracts are complex and can easily become inconsistent, which can lead to unsound reasoning. Showing that a model class can be mapped consistently to a mathematical structure proves that the model class contract itself is (relatively) consistent. In fact, as will be shown in Chapter 11, one of our case studies discovered an inconsistent specification in class `JMLObjectSet`, which is one of the most basic model classes of JML's model library.

Proving that the specification of a model class is complete relatively to a theory gives some level of confidence that the model class contains

all important properties. Failing to prove completeness is typically a sign that some properties are missing. Our case studies discovered such missing specifications. Examples will be given in Chapter 11.

These points show that proving faithfulness of mappings helps in writing better specifications for model classes by making them consistent and complete. Our approach can also be used to identify redundant parts of specifications as well as to check whether specifications marked as redundant are indeed derivable from non-redundant specifications. These capabilities further improve the quality of model-class specifications.

Remark. In the sequel, we use JML^{--} as specification language and Isabelle as the target theorem prover. However, the presented approach is applicable to any combination of specification language and theorem prover, for instance, Eiffel and Coq.

Outline. The rest of Part II is structured as follows. The remainder of this chapter introduces two running examples through which the main concepts of our approach will be presented: a JML model class and an Isabelle theory.

Chapter 9 presents the technical details of the proposed approach. In particular, the way translations are derived from specified mappings is presented as well as the proof obligations that are to be proven to complete the consistency and completeness proofs for a given mapping.

Chapter 10 discusses various aspects of our approach, such as its shortcomings and its potential for automation. Chapter 11 presents the case study we performed on model class `JMLObjectSet`.

Chapter 12 extends the approach with the handling of model classes that are mapped to inductive data types. Inductive data types come with a variety of properties such as the induction principle that can typically not be derived from first-order invariants and method specifications. Therefore, a different proof technique is required to show completeness of a model class with respect to an inductive data type. The chapter contains a second case study: the mapping of model class `JMLObjectSequence` to the inductively defined type `HOL/List`.

Finally, Chapter 13 gives an overview of related work.

```
mapped_to("Isabelle", "HOL/Set", "α set");
immutable pure model class JMLObjectSet {

    invariant (\forall JMLObjectSet s2. s2 != null ==>
                (\forall Object e1. (\forall Object e2.
                  equational_theory(this, s2, e1, e2))));

    static pure boolean
      equational_theory(JMLObjectSet s, JMLObjectSet s2,
                        nullable Object e1, nullable Object e2)
        ensures \result ==
          (s.union(s2)).has(e1) == (s.has(e1) || s2.has(e1));
        // other postconditions omitted

}
```

Figure 8.2: A snippet of the equational theory of `JMLObjectSet`

8.2 Running Examples

8.2.1 Model Class `JMLObjectSet`

Model class `JMLObjectSet` is part of JML's model library [67]. The class encodes sets of objects: it provides the usual operations of mathematical sets; equality over the set-elements is based on Java's reference equality ("`==`"). Figure 8.2 presents a small part of the *equational theory* of the class, explained below. Figure 8.3 presents those methods of the class and their specifications that are discussed in the sequel. Other methods and specification elements are omitted for brevity.[2]

Beside the two constructors presented in Figure 8.3, the class has a third one, too: JML provides a special construct to express *set comprehension*. For example, the following expression yields a set of `Integer` objects that are contained by collection `s` and whose integer values are greater than 0 [77]:

```
new JMLObjectSet { Integer i | s.has(i) && 0 < i.intValue() }
```

The syntax of JML requires that the predicate of the set comprehension is a conjunction, and the first conjunct is a method call that checks if the bound variable is a member of a collection or a model class that represents a set (*i.e.*, `JMLObjectSet` or `JMLValueSet`). Thus, model class `JMLObjectSet` represents a finite set.

Class `JMLObjectSet` is specified to be pure, thus, all its instance methods are pure. Furthermore, the class is specified to be immutable. Methods that return `JMLObjectSets` (*e.g.*, method `union`) do not mutate their receiver objects but return new instances.

[2]The presented parts of the class are split over two figures because otherwise they would not fit on one page.

```
mapped_to("Isabelle", "HOL/Set", "α set");
immutable pure model class JMLObjectSet {

   mapped_to("Isabelle", "{}");
   JMLObjectSet();

   mapped_to("Isabelle", "insert e {}");
   JMLObjectSet(nullable Object e);

   mapped_to("Isabelle", "elem : this");
   boolean has(nullable Object elem);

   mapped_to("Isabelle", "this = s2");
   boolean equals(Object s2);

   mapped_to("Isabelle", "this = {}");
   boolean isEmpty();

   int int_size();

   mapped_to("Isabelle", "this <= s2");
   boolean isSubset(JMLObjectSet s2);

   mapped_to("Isabelle", "this < s2");
   boolean isProperSubset(JMLObjectSet s2);

   mapped_to("Isabelle", "SOME x. x : this");
   nullable Object choose();

   mapped_to("Isabelle", "insert elem this");
   JMLObjectSet insert(nullable Object elem)
     ensures (\forall Object e. \result.has(e) ==
                (this.has(e) || e == elem));

   mapped_to("Isabelle", "this - (insert elem {})");
   JMLObjectSet remove(nullable Object elem);

   mapped_to("Isabelle", "this Un s2");
   JMLObjectSet union(JMLObjectSet s2)
      ensures (\forall Object e. \result.has(e) ==
                 (this.has(e) || s2.has(e)));

   mapped_to("Isabelle", "this - s2");
   JMLObjectSet difference(JMLObjectSet s2);

   mapped_to("Isabelle", "Pow this");
   JMLObjectSet powerSet();

   // other methods omitted
}
```

Figure 8.3: Signatures and mappings of `JMLObjectSet`'s methods

The class is specified by method specifications and an invariant. Two sample method specifications are given in Figure 8.3 for methods `insert` and `union`. The proposed mapping of the class and its methods to one of Isabelle's set structures is given by `mapped_to` clauses that we explain in Chapter 9.

In Figure 8.2, the invariant expresses that method `equational_theory` has to return true in every visible state for all non-null `JMLObjectSet` instances **`this`** and `s2`, and objects `e1` and `e2`. Method `equational_theory` is a static pure method and has a large method specification that contains equations written in the style of algebraic laws. As a result, the invariant of the model class prescribes that all equations specified by the method specification of `equational_theory` have to hold in all visible states. For brevity, Figure 8.2 shows only a sample equation defining method `union`.

This way of writing invariants is typical for model classes and such invariants are commonly referred to as the *equational theory* of the class. We will use this terminology in the sequel, too. Furthermore, we will use the term "invariant" when referring to an equation prescribed by the specification of method `equational_theory`, for instance, to the equation that defines method `union`.

We follow the proposal of Leavens *et al.* [75] and Charles [29], and consider model classes to be (implicitly) final and unrelated to Java's type hierarchy rooted in class `Object`. This prevents problems related to inheritance, method overriding, and dynamic dispatch. In the realm of model classes, these restrictions seem acceptable since model classes are supposed to describe elementary mathematical concepts and to be used only for specification purposes.

On the other hand, the restrictions result in the limitation that, for instance, a set of sets or a list of lists cannot be represented. Consider expression `powerSet(s).remove(s)` and `powerSet(s).choose()` for some `JMLObjectSet`-instance `s`. According to our understanding of set theory, the expressions are meaningful. However, if model class `JMLObjectSet` is not a subtype of `Object`, then the expressions do not typecheck: method `remove` takes an `Object` and `choose` returns an `Object`, which is not compatible with the `JMLObjectSet`-instance that the methods take and return, respectively.

A solution for this problem would be to follow the approach that theorem provers typically apply, namely, to use a type variable to denote the type of elements of a structure (*e.g.*, α in Isabelle). This could be realized by the use of generics with type parameters whose upper bound is not fixed in type `Object`. Given such a type system, methods like `powerSet` could be mapped to theorem provers that support type variables.

In the sequel, we make the following assumption, which can be checked by the approach described in Chapter 6, and which will allow us to concentrate on the *normal behavior* specification cases of methods.

Assumption 8.1. *Model specifications and specifications that rely on model classes are well-defined.*

The well-definedness of such specifications can be checked by the technique presented in Part I.

8.2.2 Isabelle/HOL and Theory `HOL/Set`

Isabelle [106] is a generic interactive LCF-style theorem prover. It is generic in that it provides a meta-logic, which allows one to implement different logical formalisms, for instance, FOL (first-order logic) and ZF (Zermelo-Fraenkel set theory). In the sequel, we use Isabelle/HOL [106, 107], which is the specialization of Isabelle for the logical system of higher-order logic. In what follows, we will interchangeably use the term "Isabelle" to refer both to Isabelle, the theorem prover, and to Isabelle/HOL, the logical system.

Isabelle is interactive, that is, users have full control over the process of proving theorems. However, several automatic decision procedures and tableaux provers (collectively called *tactics*) are available, which significantly simplify and accelerate the proof process. Two of the often used tactics are *simp*, which is based on term rewriting; and *auto*, which is a tableaux prover.

Isabelle is an LCF-style theorem prover, that is, it is based on a small logical core [48]. Everything else is supposed to be defined on top of this core by conservative extensions, which ensure its logical consistency. Although Isabelle allows users to state axioms, it is strongly discouraged due to the risk of making the specification inconsistent.

When working with Isabelle, users write theories. Theories typically consist of type and function-symbol declarations, and of definitions and theorems over these types and symbols.

In order to introduce the Isabelle constructs that are necessary for the understanding of the subsequent chapters, we briefly present a tiny part of theory `HOL/Set`. The relevant parts of the theory are shown in Figure 8.4.

Remark. The presented theory is not part of the latest Isabelle 2008 distribution, but of the previous, Isabelle 2005 distribution. The reason is that the former introduces several dependencies to other theories of the library, which improves the theory library, but which would make the presentation of theory `HOL/Set` more difficult.

The theory begins by importing another theory `HOL/LOrder`, the theory of lattice orders. Thereby, all symbols, definitions, axioms, and theorems that are available in theory `HOL/LOrder` are also available in `HOL/Set`. Next, the new type α `set` is introduced, where α is a type variable that gives rise to polymorphic types [106].

```
theory Set
imports LOrder
begin

typedecl α set

consts
  "{}"          :: "α set"
  insert        :: "α  ⇒ α set ⇒ α set"
  Collect       :: "(α  ⇒ bool) ⇒ α set"
  Un            :: "α set ⇒ α set ⇒ α set"
  ":"           :: "α  ⇒ α set ⇒ bool"

translations
  "{x. P}"     == "Collect (λ x. P)"

axioms
  mem_Collect_eq: "(a : {x. P(x)}) = P(a)"
  Collect_mem_eq: "{x. x:A} = A"

defs (overloaded)
  subset_def:  "A <= B     == ∀ x. x:A ⇒ x:B"
  psubset_def: "A < B      == A <= B & ~A=B"
  set_diff_def: "A - B     == {x. x:A & ~x:B}"

defs
  Un_def:      "A Un B     == {x. x:A | x:B}"
  empty_def:   "{}         == {x. False}"
  insert_def:  "insert a B == {x. x=a} Un B"

lemma subsetI: "(∀ x. x:A ⇒ x:B) ⇒ A <= B"
  by (simp add: subset_def)

end
```

Figure 8.4: A snippet of Isabelle's HOL/Set theory

Keyword **consts** declares five new function symbols for the empty set, element insertion, set comprehension, union (`Un`), and set membership, respectively. Union and set membership are defined to be infix operators—not shown in the figure to avoid unnecessary syntactic clutter.[3]

The constant declaration is followed by the definition of syntactic sugar for set comprehension and the axiomatization of set comprehension. The axioms are followed by definitions; the first group defines the semantics of symbols that are not introduced by theory `HOL/Set` but are already available in the imported theory `HOL/LOrder`. These are subset, proper subset, and set difference. The second group defines the newly introduced symbols—except of set membership and set comprehension.

Finally, a theorem is stated (introduced by keyword **lemma**) and proven using the *simp* tactic, which uses a pre-defined set of axioms, definitions, and theorems. For the proof of the theorem, this set is extended by definition `subset_def`, the definition of subset given in the theory.

[3]We also omitted other, rather cryptic parts to ease understanding. As a result, the presented specification does not parse with Isabelle.

Chapter 9

Faithful Mapping

In this chapter, we present our approach for proving that the mapping of a model class M to a mathematical structure S is faithful. That is, we show that there is a standard interpretation of the theory formed by the specification of M in the theory of S; and vice versa.

The process of proving this correspondence consists of three stages. In the first stage, we specify the mapping of M to S using `mapped_to` clauses. In the second stage, we deduce a standard translation from the theories of M and S based on the specified mappings, and attempt to prove that the translation yields a standard interpretation. In the third stage, we attempt to do the same in the reverse direction. In this chapter, we assume that the definition of structure S is non-inductive. The handling of inductive structures is described in Chapter 12.

Remark. The approach presented in this thesis does not handle ghost fields of model classes. Ghost fields can be handled by mapping a model class M with n ghost fields to an $n+1$-tuple, where the first component represents the structure for M and the other components represent the state of the ghost fields [99]. Omitting ghost fields allows us to map a model class M directly to some structure S and to avoid cluttering up the proof obligations with projections of tuples.

Contexts and Function $\overline{\gamma}$. In the second and third stages of the faithfulness proof, proofs are carried out in two different contexts. In the second stage, the context is that of S. In the third stage, the context is conceptually that of M, however, that context (essentially JML^{--}) is not adequate for formal proof. Therefore, another context, denoted by $\widehat{M}$, is used that encodes the specification of M in a theorem prover, thereby allowing one to carry out the proofs of the third stage using a formal system, like Isabelle or Coq.

The way context $\widehat{M}$ is generated will be described in Section 9.3.3. However, before that, we will make use of function $\overline{\gamma}$ that encodes specification

expressions in context $\widehat{M}$. The function takes a JML^{--} expression and yields a first-order term or formula:[1]

$$\overline{\gamma} : Expr \rightarrow Term_{\widehat{M}}$$

The function is defined the same way as γ in Figure 4.3 on page 50, except that $\overline{\gamma}$: (1) does not take heap arguments, (2) does not handle field accesses, and (3) does not restrict the scope of quantification to allocated objects:

$\overline{\gamma}(\texttt{(\forall}\ T\ x.\ E\texttt{)}) \triangleq \forall\, x.\, \overline{\gamma}(E)$ and
$\overline{\gamma}(\texttt{(\exists}\ T\ x.\ E\texttt{)}) \triangleq \exists\, x.\, \overline{\gamma}(E)$.

Note that for method calls, function $\overline{\gamma}$ yields the uninterpreted "hat"-functions that were introduced in Part I, just with no heap argument. That is, $\overline{\gamma}(E\texttt{.m(}F\texttt{)}) \triangleq \widehat{m}(\overline{\gamma}(E), \overline{\gamma}(F))$.

9.1 Specifying the Mapping

In the first stage, one has to decide how to map model class M. That is, one has to specify: (1) to what structure S is the model class mapped, and (2) to which symbols of S are the methods of the model class mapped.

9.1.1 Specifying the Mapping of Classes and Methods

The mapping of a model class is specified by a `mapped_to` clause attached to the class. The first argument of the clause specifies the target theorem prover, the second the target theory, and the third the specific type (if any) in the theory to which the model class is mapped. Only one clause per target theorem prover may be specified.

Figure 8.3 on page 121 shows a possible mapping of class `JMLObjectSet`: it is mapped to type α *set* in the `HOL/Set` theory of Isabelle.

The mapping of a method is specified by a `mapped_to` clause attached to it. The first argument of the clause is again the target prover, the second specifies the term to which the method is mapped in the target theory. The term may only mention logical and nonlogical symbols of the target theory and parameters (including the explicit receiver) of the specified method. The argument must contain a term that is well-typed, otherwise the translation of specification expressions fails. Only one clause per target theorem prover may be specified.

In Figure 8.3, `JMLObjectSet`'s `has` method is mapped to the term *elem* : *this*, meaning that the method corresponds to Isabelle's set membership operator ":".

[1] The function may yield a *Formula*, too. For brevity, we do not distinguish the two cases in the signature of $\overline{\gamma}$ and other functions introduced in the rest of this chapter.

We permit one to write arbitrarily complex terms, which allows our approach to support method with functionality that is not directly supported by the target prover. This flexibility is necessary to handle, for example, `JMLObjectSet`'s `remove` method, which removes a single element of a set. Theory `HOL/Set` does not provide a corresponding operation but provides set difference, which allows one to express the meaning of method `remove`. Thus, if our approach allowed only direct correspondence between methods and function symbols, then this simple operation could not be handled with the chosen mapping for class `JMLObjectSet`.

Remark. Note that `mapped_to` clauses are used only for the purpose of static verification. For runtime assertion checking, the clause has no meaning and can be ignored.

Remark. In Section 2.3, the concept of theory interpretation was presented for first-order languages. This seems to be reasonable since JML^{--} specifications are first-order. However, above, class `JMLObjectSet` was proposed to be mapped to a theory of higher-order logic. Thus, applying the theory over first-order languages does not seem adequate for the faithfulness proof and, in particular for the completeness proof, where the source of the translation is a higher-order context.

As we will see in later chapters, our approach is to apply first-order translation and, whenever needed, approximate the translation of higher-order properties with first-order formulas. The reason for this approach is that the interpretation of higher-order theories is significantly more difficult than that of first-order theories [42], and would be mostly an overkill for the handling of model classes.

Supporting Multiple Provers. One might want to provide mappings for a model class to different theorem provers at the same time. For example, besides Isabelle, one might want to specify the mapping of class `JMLObjectSet` to PVS and Coq, too. Therefore, multiple `mapped_to` clauses may be attached to the model class and its methods. For example, model class `JMLObjectSet` and its `has` method could be mapped to Coq as follows:

```
mapped_to("Coq", "Coq.Sets.Classical_sets", "Ensemble");

mapped_to("Coq", "In this elem");
```

Allowing mappings to different theorem provers is needed since different provers provide different theories with different symbols and corresponding semantics. As a consequence, the faithfulness proof has to be carried out in every target prover specified in `mapped_to` clauses.

Remark. Mappings need not be specified by programmers who are typically not familiar with theorem provers and their theories. Mappings can be specified by the author of a model class or by the team that performs the verification. The same applies to the proof of faithfulness, which typically requires experience in theorem proving.

Function ν. Following the notation of Farmer [42], we use function ν to map methods to term and formula functions of the target theory. As `mapped_to` clauses contain exactly that information, function ν essentially captures the content of the clauses. For instance, for model class `JMLObjectSet` we have:

$$\begin{aligned} \nu(\texttt{has}) &\equiv \lambda\{this, elem.\ elem : this\} \\ \nu(\texttt{remove}) &\equiv \lambda\{this, elem.\ this - (insert\ elem\ \{\})\} \end{aligned}$$

Remark. As a second argument, ν should take the name of the target theorem prover as the mapping of a method may be different for different provers. For simplicity, we omit this argument as the target prover is always going to be Isabelle in the sequel.

9.1.2 Defining the Universe

Recall from Section 2.3 that a standard translation Φ has two components, a map ν and the *universe predicate* U. U is a closed unary predicate of the target theory that takes a value that denotes an element of the target structure. The predicate is defined such that it yields true if and only if its argument is in the scope of translation Φ. We will denote the universe used during the consistency proof by U_S and the one used during the completeness proof by $U_{\widehat{M}}$. When proving the faithfulness of a given mapping, predicates U_S and $U_{\widehat{M}}$ correspond to the same universe, just the contexts in which they are expressed differ.

The main use of the universe predicate is to "relativize" the scope of quantifiers (see *translation of E via Φ* on page 27). The need for relativization is demonstrated by the following example.

Example 9.1. Consider model class `SmallSet` in Figure 9.1. The class represents a set that is either the empty set or a singleton set. The class has methods for creating such sets, for removing an element from the set, and for determining set membership. The class is specified to be mapped to Isabelle's `HOL/Set` theory. For brevity, only one specification element is presented, others are omitted.

The specification of method `remove` expresses that removing an element of the set yields the empty set. Clearly, this property holds for instances of `SmallSet`; however, it does not hold for sets in general. Thus, if the specification element was translated to Isabelle without taking into account the special universe of `SmallSet`-instances, then the specification of `remove`

```
mapped_to("Isabelle", "HOL/Set", "α set");
immutable pure model class SmallSet {

    mapped_to("Isabelle","{}");
    constructing
    SmallSet();

    mapped_to("Isabelle","insert e {}");
    constructing
    SmallSet(nullable Object e);

    mapped_to("Isabelle","this - (insert e {})");
    SmallSet remove(nullable Object e)
      ensures this.has(e) ==> \result.equals(SmallSet());

    mapped_to("Isabelle","e : this");
    boolean has(nullable Object e);

    // other specifications omitted
}
```

Figure 9.1: Model class `SmallSet`, a set with a limited universe

would get rejected while proving consistency of the mapping. Similarly, when proving completeness of the mapping, certain definitions of the target theory would probably not be provable.

As a consequence, the faithfulness of mapping class `SmallSet` to Isabelle's set theory would not be provable. However, `SmallSet` does represent a proper set, just with a limited universe.

The problem can be resolved by defining a proper predicate for the class that restricts the universe of `SmallSet`-instances. Namely, the predicate (shown in Example 9.2) has to express that every instance of the class is equivalent to the value that one of the two constructors possibly yield. Consequently, the property expressed by the postcondition of `remove` becomes provable in the target theory; and while proving completeness, set-properties only need to be proven with respect to the limited universe, too. Thereby, the rejection of the actually faithful mapping is prevented. □

To allow users to specify the set of methods that should form the universe predicate of a model class, we introduce the **`constructing`** modifier for model-class methods. For instance, in Figure 9.1, the two constructors are marked as **`constructing`**.

Given a model class with l methods marked as **`constructing`**, the resulting universe predicate will consist of a disjunction with l disjuncts. Assuming, for simplicity, that parameters of the enclosing type appear before the parameters of other types in the signatures of methods, the effect of marking method m with one implicit and n explicit parameters, and pre-

condition P is that the universe predicate $U_{\widehat{M}}(x)$ will contain a disjunct of the form:

$$\exists\, s, e_1, \ldots, e_n.\ U_{\widehat{M}}(s) \wedge U_{\widehat{M}}(e_1) \wedge \ldots \wedge U_{\widehat{M}}(e_k) \wedge \overline{\gamma}(P) \wedge \widehat{equals}(x, \widehat{m}(s, e_1, \ldots, e_n))$$

where $k \leq n$ and parameters $e_1, \ldots, e_k$ are of the enclosing type, while the others, if any, are not.

The construction of predicate U_S is analogous. The only difference is that the property is expressed in the context of the target theory:

$$\exists\, s, e_1, \ldots, e_n.\ U_S(s) \wedge U_S(e_1) \wedge \ldots \wedge U_S(e_k) \wedge \Phi_M(P) \wedge \Phi_M(x\texttt{.equals(}s\texttt{.m(}e_1, \ldots, e_n\texttt{))})$$

where Φ_M is the translation function between JML^{--} expressions and the target context. The function is precisely defined in the next section.

Example 9.2. Having specified the two constructors of class `SmallSet` being **`constructing`** in Figure 9.1, we get the following universe predicates in the context of the model class and the target theory, respectively:

$$U_{\widehat{M}}(x) \triangleq \widehat{equals}(x, \widehat{SmallSet}()) \vee (\exists\, e.\ \widehat{equals}(x, \widehat{SmallSet}(e)))$$

$$U_S(x) \triangleq x = \{\} \vee (\exists\, e.\ x = (insert\ e\ \{\})) \tag{9.1}$$

□

Note that the universe predicate is defined inductively, which can make faithfulness proofs difficult. The properties that need to be shown during the consistency proof have to be proven by induction over the structure of terms that denote instances of the target structure. Analogously, when proving completeness, the induction goes over the JML^{--} expressions that denote instances of the model class. For instance, given the following universe predicate for class `JMLObjectSet`:

$$U_S(x) \triangleq x = \{\} \vee (\exists\, s, e.\ U_S(s) \wedge x = (insert\ e\ s))$$

one would need to use an induction hypothesis that states that the property at hand holds for some instance y from which x can be yielded by an application of *insert*.

If no method is marked as **`constructing`** then the default value of the universe predicate is *true* and can be omitted in the translation. As mentioned on page 27, this is the case when the source and the target universe

is the same. Therefore, when a model class has the standard universe of the structure that it represents, then the **constructing** modifier is typically not needed. This often happens in practice, for instance, this is the case for class JMLObjectSet, as discussed in Chapter 11.

Remark. Since the soundness of theory interpretations does not depend on predicate U, there is no need to check that the set of methods marked as **constructing** is minimal, or that indeed the whole universe of the enclosing class can be constructed by the marked methods. Of course, a bad choice of the predicate may lead to a failing faithfulness proof of two semantically equivalent structures, as would be the case for class SmallSet, if the **constructing** modifier was omitted from Figure 9.1.

9.2 Proving Consistency of a Model Class

In the second stage of the faithfulness proof, we prove consistency of the mapping, that is, we show that there is a standard interpretation of M's theory in S's theory. Therefore, we need to translate the method specifications and invariants of M in the context of S based on the mapped_to clauses, and prove the resulting formulas using the properties of S. To do so, first the translation function Φ (presented on page 27) has to be adapted to the context of JML^{--} specifications. The resulting translation function will be denoted by Φ_M.

Definition of Φ_M. The function takes a JML^{--} expression and yields a term or formula in the target theory S:

$$\Phi_M : Expr \rightarrow Term_S$$

The definition of the function is presented in Figure 9.2. Note that the definition is similar to that of γ in Figure 4.3 on page 50, except that (1) it does not take heap arguments, (2) method calls, **result**, and **this** in postconditions of constructors are translated according to the specified mapped_to clauses, and (3) quantifiers are relativized by the universe predicate.

The definition of Φ_M is rather straightforward. For simplicity, the definition for method and constructor calls, for keyword **result**, and for keyword **this** in the postcondition of constructors is presented for methods and constructors with one explicit parameter p. Note that terms $\nu(\mathtt{m})(this, p)$ and $\nu(\mathtt{C})(p)$ denote the terms that are defined by the mapped_to clause of the corresponding method and constructor. The old-construct is "ignored" by the translation since the construct is not meaningful in pure-method specifications.

$$
\begin{array}{lcll}
\Phi_M(E \circledast F) & \triangleq & \Phi_M(E)\ FOL(\circledast)\ \Phi_M(F), & \text{where } \circledast \text{ and } FOL \\
 & & & \text{are defined in Figure 2.1 and 4.4, resp.} \\
\Phi_M(\texttt{!}E) & \triangleq & \neg\, \Phi_M(E) & \\
\Phi_M(E\texttt{.m(}F\texttt{)}) & \triangleq & \nu(\texttt{m})(\Phi_M(E), \Phi_M(F)) & \\
\Phi_M(\textbf{\texttt{new}}\ \texttt{C(}E\texttt{)}) & \triangleq & \nu(\texttt{C})(\Phi_M(E)) & \\
\Phi_M(\texttt{\textbackslash}\textbf{\texttt{result}}) & \triangleq & \nu(\texttt{m})(\mathit{this}, p), & \text{where m is the enclosing method} \\
\Phi_M(\textbf{\texttt{this}}) & \triangleq & \nu(\texttt{C})(p)\,, & \text{if } \textbf{\texttt{this}} \text{ occurs in the post-} \\
 & & & \text{condition of constructor C} \\
\Phi_M(\texttt{v}) & \triangleq & v\,, & \text{if v is a parameter or literal} \\
 & & & \text{other than } \texttt{\textbackslash}\textbf{\texttt{result}} \text{ and } \textbf{\texttt{this}} \text{ in} \\
 & & & \text{postconditions of constructors} \\
\Phi_M(\texttt{\textbackslash}\textbf{\texttt{old}}\texttt{(}E\texttt{)}) & \triangleq & \Phi_M(E) & \\
\Phi_M(\texttt{(\textbackslash}\textbf{\texttt{forall}}\ T\ x\texttt{.}\ E\texttt{)}) & \triangleq & \forall\, x.\ \boxed{U_S(x)\ \Rightarrow}\ \Phi_M(E) & \\
\Phi_M(\texttt{(\textbackslash}\textbf{\texttt{exists}}\ T\ x\texttt{.}\ E\texttt{)}) & \triangleq & \exists\, x.\ \boxed{U_S(x)\ \wedge}\ \Phi_M(E) &
\end{array}
$$

where the shaded parts are added only if the quantified variable is of a model type.

Figure 9.2: Definition of translation Φ_M

9.2.1 Proving that Φ_M is a Standard Interpretation

It remains to prove that the standard translation Φ_M is a standard interpretation of M's theory in S's theory. To do so, we need to prove that the three obligations presented on page 27 hold. In the remainder of this section, we show how these obligations are to be enforced.

Axiom Obligation. Recall that the axiom obligation requires that the translation of every axiom of M is a theorem of S. The "axioms" of a model class are its invariants and method specifications. Their translation is straightforward, only the free variables have to be bound by universal quantifiers since these quantifications are implicit in invariants and method specifications:

- an invariant Inv of some class C is translated to the formula:

  $$\Phi_M(\texttt{(}\textbf{\texttt{\textbackslash forall}}\ C\ \textbf{\texttt{this}}\texttt{.}\ \mathit{Inv}\texttt{)})$$

  or equivalently to:

  $$\forall\, \mathit{this}.\ U_S(\mathit{this}) \Rightarrow \Phi_M(\mathit{Inv}).$$

- the specification of a method of class C with one explicit parameter p of type T, precondition P, and postcondition Q is translated to:

Φ_M(**(\forall** C **this. (\forall** T p. P **==>** Q**))**)

which is equivalent to:

$\forall\ this, p.\ U_S(this) \Rightarrow$ $U_S(p) \Rightarrow$ $\Phi_M(P$ **==>** $Q)$.

where the shaded part is only added if p is of a model type.

These formulas are turned into lemmas and have to be proved in the target theory.

Remark. Note that invariants are not conjoined to pre- and postconditions. This is because invariants of model classes do not restrict the state space of their instances, but rather give equational laws about their operations. These laws are dealt with separately when they are turned into lemmas.

Remark. Similarly to the encoding of pure methods described in Part I, the translation of model specifications considers only method specifications that prescribe *normal behavior*. This is justified by Assumption 8.1 on page 123, namely, that model and client specifications are well-defined.

Example 9.3. Consider the specification of method `insert` in model class `JMLObjectSet`, given in Figure 8.3 on page 121. The resulting formula is:

$$\forall\ this, elem.\ U_S(this)\ \Rightarrow\ \Phi_M(P \texttt{ ==> } Q)$$

where P is **true** and Q is:

```
(\forall Object e. (\result.has(e) == (this.has(e) || e == elem)))
```

The most interesting part of the application of function Φ_M is that it translates **\result** to term *insert elem this*, and that it translates the two calls to method **has** to applications of the set membership operator, as prescribed by the `mapped_to` clause of the method. The resulting formula is the following:

$$\forall\ this, elem.\ U_S(this)\ \Rightarrow \\ (\forall e.\ (e : (insert\ elem\ this)) = ((e : this)\ \vee\ (e = elem)))$$

The formula is turned into a lemma in Isabelle's `HOL/Set` theory. The lemma can be proved automatically by the *auto* tactic, which is not surprising as theorem provers like Isabelle are typically well-equipped with theorems over their built-in structures. Note that it does not matter how predicate U_S is defined, because the consequence of the implication holds in the chosen Isabelle theory. □

Universe Nonemptiness Obligation. Recall that the obligation requires that the universe of the translation is nonempty: $\exists\, x.\ U_S(x)$. This is usually a trivial proof obligation. For instance, in Example 9.1 on page 130, picking $\{\}$ for x trivially discharges the obligation. Despite the existential quantification, the proof obligation is automatically proved by Isabelle's *auto* tactic.

Note also that the obligation trivially holds if the universe predicate is defined to be the trivial $U_S(x) \triangleq (x = x)$.

Function Symbol Obligation. Recall that the obligation requires that for each symbol f of theory T, an application of f with arguments from the universe is translated to a term that denotes a value of the universe.

When applying the obligation for model-class methods, the only difference is that specified preconditions have to be taken into account as well. Assuming, for simplicity, that parameters of model types appear before the parameters of other types in the signatures of model classes, we have to prove for each model method m with n explicit parameters and precondition P that the following holds in the target theory:

$$\begin{aligned}\forall\, t, x_1, \ldots, x_n.\ & U_S(t) \Rightarrow U_S(x_1) \Rightarrow \ldots \Rightarrow U_S(x_k) \Rightarrow \\ & \Phi_M(P(t, x_1, \ldots, x_n)) \Rightarrow U_S(\Phi_M(t\texttt{.}m\texttt{(}x_1, \ldots, x_n\texttt{)}))\end{aligned} \tag{9.2}$$

where $k \leq n$ and parameters $x_1, \ldots, x_k$ are of model types, while the others, if any, are not. The proof obligations for the constructors of a model class are constructed analogously.

Example 9.4. We present a non-trivial instance of the function symbol obligation. The property to show for `SmallSet`'s `remove` method is:

$$\begin{aligned}&\forall\, t, x.\ U_S(t) \Rightarrow U_S(\Phi_M(t\texttt{.remove(}x\texttt{)})) \ \equiv \\ &\forall\, t, x.\ U_S(t) \Rightarrow U_S(t - (\mathit{insert}\ x\ \{\}))\end{aligned}$$

where U_S is the predicate defined by (9.1) on page 132. The formula can be proven by Isabelle's *auto* tactic after unfolding both occurrences of U_S. □

The second stage of the faithfulness proof is successfully completed if all three obligations can be proven. Based on Theorem 2.2 (presented on page 28), we can then conclude that the specification of the model class at hand is consistent provided that the target theory is consistent.

9.2.2 Consistency and Sound Verification

Having proved consistency of the specification of a model class guarantees that the specification is free from contradictions. Consequently, it can be

safely used for reasoning about client code. However, having proved consistency of the specification is not sufficient for the use of `mapped_to` clauses for verification purposes, because the symbol to which a method is mapped to may be specified to possess properties that the method does not, thereby leading to unexpected results during the verification of client code.

Example 9.5. Suppose class `JMLObjectSet` had a method `bogusUnion` that was mapped to Isabelle's set union operation and was specified as follows:

```
mapped_to("Isabelle", "this Un s2");
JMLObjectSet bogusUnion(JMLObjectSet s2)
  ensures !this.isEmpty() ==> !\result.isEmpty();
```

The axiom obligation can be proved trivially for the specification since the specified property holds for Isabelle's union operator *Un*, too:[2]

$$\begin{array}{ll} \Phi_M(\texttt{!this.isEmpty() ==> !\result.isEmpty()}) & \equiv \\ \neg(this = \{\}) \;\Rightarrow\; \neg((this\ Un\ s2) = \{\}) & \equiv \\ true & \end{array}$$

However, consider the following boolean expression:

```
new JMLObjectSet().bogusUnion(new JMLObjectSet(e)).has(e)
```

The specification of `bogusUnion` would not allow us to determine whether the expression holds or not, however, the translation of the expression yields the formula $e : (\{\}\ Un\ (insert\ e\ \{\}))$, which is trivially provable in Isabelle. Thus, it would be unsound for a program verifier to map method `bogusUnion` to Isabelle's *Un* symbol, since we would allow more properties to be proven after the mapping than we could prove before the mapping, using only the model-class specifications.

Furthermore, the symbols to which method `bogusUnion` is mapped in other theorem provers might be defined differently. Thus, the outcome of verification attempts might be theorem-prover dependent. Runtime assertion checking might also lead to unexpected results, because the specification of the method allows, for instance, the implementation **`return this`**, which clearly violates the properties of set union. □

Intuitively, the problem is that the specification of `bogusUnion` would describe only one particular property of mathematical union of sets. However, after having mapped `bogusUnion` to Isabelle's *Un* symbol, the method would be endowed with all the additional properties that symbol *Un* has. For the faithful mapping of `bogusUnion`, we would need to show that the method indeed possesses these endowed properties.

This example demonstrates that proving completeness of a model class with respect to a theory does not just show that the specification of the class

[2]For brevity, quantification and the universe predicate are omitted from the translation.

is strong enough relative to the theory, but is crucial for the sound use of `mapped_to` clauses during the verification of client code.

9.3 Proving Completeness of a Model Class

In the third stage of the faithfulness proof, we prove completeness of the mapping, that is, we show that there is a standard interpretation of S's theory in M's theory. Therefore, we need to translate the specifications of S (typically axioms and definitions) in the context of M. As before, we first need to define a translation function, denoted by Φ_S, and then prove the three obligations.

9.3.1 Issues of the Reverse Mappings

It is crucial to note that the completeness proof should not only show that the model class provides all the functionality that the Isabelle structure provides. It should also show that related model-class methods and functions indeed correspond to each other. For instance, it is not sufficient to prove that the specification of method `union` is complete with respect to the properties of function *Un*: One additionally needs to prove that the specification of a method such as `bogusUnion` is also complete with respect to *Un*, because calls to `bogusUnion` in specifications would also be translated to applications of *Un*.

This means that translation Φ_S may not be an arbitrary standard translation for which we can show that it is a standard interpretation. Instead, Φ_S should be one that is derived from the mapping that is specified by the `mapped_to` clauses. That is, we need a way to *reverse* specified mappings. In the following, we show that this is not trivial because symbols of S might be mapped to by *multiple* methods, and some of the reverse mappings are only valid under certain *conditions*.

Consider symbol *insert* of theory `HOL/Set`, which is mapped to both by method `insert` and by constructor `JMLObjectSet(e)`:[3]

$$\begin{array}{lcl} \nu(\texttt{insert})(this, e) & \equiv & insert\ e\ this \\ \nu(\texttt{JMLObjectSet(e)})(e) & \equiv & insert\ e\ \{\} \end{array}$$

Therefore, as argued above, the translation of a formula that contains an application of the symbol should consider mapping the symbol both to the method and to the constructor. Although this seems to be doable by defining translation Φ_S such that all possible reverse mappings of a symbol must be taken into account by a case split, clearly, such a translation would not be standard anymore (*cf.* Section 2.3).

[3] In the sequel, we will write `JMLObjectSet(e)` to refer to the constructor with one parameter even when only a method or constructor name is expected, like the argument of function ν.

Furthermore, the translation of the general term *insert x Y* to the constructor is only valid under the condition that Y corresponds to the empty set. This condition would need to be added to the translated formula, which again shows that the translation would not be standard.

In the sequel, we demonstrate through an example that the problem is not merely that the resulting reverse translation would not be standard. We show that the conditions under which certain mappings are valid may alter the semantics and satisfiability of the original formula.

Example 9.6. It is well-known that a condition over a universally bound variable has to be added as the premise of an implication, otherwise the condition has to be added as a conjunct. But what if a condition contains both existentially and universally quantified variables?

Consider the following theorem of Isabelle's `HOL/Set` theory:

$$\forall\, A, x. \exists\, B.\ (\mathit{insert}\ x\ (A\ \mathit{Un}\ B)) = (\mathit{insert}\ x\ A)\ \mathit{Un}\ B$$

Let us look at the translation when the first application of *insert* is mapped to constructor `JMLObjectSet(e)`. As discussed above, the translation of the term *insert x Y* to the constructor yields the condition that Y corresponds to the empty set. Therefore, the translation of the term *insert x* (*A Un B*) yields the condition that the union of A and B corresponds to the empty set. The condition contains both a universally and an existentially quantified variable, thus we can add the condition neither as a premise, nor as a conjunct.

In the first case, the shape of the resulting formula would be:

$$\forall\, A, x. \exists\, B.\ \widehat{\mathit{equals}}(\widehat{\mathit{union}}(A, B), \widehat{\mathit{JMLObjectSet}}())\ \Rightarrow\ \ldots$$

The problem is that one could pick a non-empty B (*e.g.*, $\mathit{JMLObjectSet}(x)$), thereby making the premise false, and the whole implication true. Thus, the resulting formula would be trivially provable and we would not ensure that the contracts of the model class are strong enough to express the property encoded by the theorem.

In the second case, the shape of the resulting formula would be:

$$\forall\, A, x. \exists\, B.\ \widehat{\mathit{equals}}(\widehat{\mathit{union}}(A, B), \widehat{\mathit{JMLObjectSet}}())\ \wedge\ \ldots$$

The formula is unprovable, as the conjunct does not hold for every A. □

These considerations point out that defining the general reverse translation of Φ_M is rather complex. In particular, due to multiple and conditional translations: (1) the translation would not be standard anymore and (2) the translation would considerably change the structure of translated formulas. Therefore, it would be difficult to reason that the resulting translation is indeed the one we are looking for.

9.3.2 Our Pragmatic Approach

To resolve the problem, we take a pragmatic approach and pose a requirement on the user-defined mappings. In practice, the requirement typically does not constrain the way model classes may be written and mapped, but it ensures that the reverse translation of Φ_M is a standard translation and can be easily derived from Φ_M.

However, just as there is no free lunch, there is no free standard reverse translation: besides the requirement, a number of proof obligations are posed on the methods of the model class at hand.

In the remainder of this section, we formalize the requirement, the reverse translation Φ_S, and the necessary proof obligations.

Requirement. The requirement we pose on specified `mapped_to` clauses is that each symbol of S should be mapped to by at least one method *unconditionally*. Formally:

> For each n-ary function and predicate symbol f of S and variables $x_1, \ldots, x_n$ there is at least one method m or constructor C of M, and expressions $e_1, \ldots, e_k$ with free variables $x_1, \ldots, x_n$ such that either $\Phi_M(e_1.m(e_2, \ldots, e_k)) = f(x_1, \ldots, x_n)$ or
> $$\Phi_M(\texttt{new}\ C(e_1, e_2, \ldots, e_k)) = f(x_1, \ldots, x_n) \text{ holds.} \tag{9.3}$$

Although the requirement does not hold for arbitrary translations, it typically holds for model classes. Conditional mappings are typically needed when a model class offers methods that are redundant in the sense that they are equivalent to some compound expression consisting of calls to more basic methods. For instance, `JMLObjectSet(e)` is equivalent to creating an empty set and inserting element `e` into it. Similarly, method `remove` is equivalent to set difference with a singleton set as second argument. Such methods make the use of model classes more convenient, but their functionalities are usually not directly available in the corresponding structures.

The requirement would not hold, for instance, if class `JMLObjectSet` provided method `remove`, but not method `difference`. However, since set difference is a more fundamental set operation than removing a single element, the requirement is likely to hold when designing a model class that represents a set.

Definition of Φ_S. Given the above requirement, the reverse translation can be easily defined. It is a transformer between terms and formulas of S and $\widehat{M}$:

$$\Phi_S : Terms_S \rightarrow Term_{\widehat{M}}$$

The definition of translation Φ_S is presented in Figure 9.3. Note that it is almost identical to Farmer's translation function Φ (see page 27). The main difference is in the translation of function and predicate symbols, which

$$
\begin{array}{lcl}
\Phi_S(Var) & \triangleq & Var \\
\Phi_S(f(t_1,\ldots,t_n)) & \triangleq & \begin{cases}
\overline{\gamma}(e_1\texttt{.m(}e_2,\ldots,e_k\texttt{)}), \\
\qquad \text{if there is a method } \texttt{m} \text{ and} \\
\qquad \text{expressions } e_1,\ldots,e_k \text{ such that:} \\
\qquad \Phi_M(e_1\texttt{.m(}e_2,\ldots,e_k\texttt{)}) = f(t_1,\ldots,t_n) \\
\overline{\gamma}(\textbf{new } \texttt{C(}e_1,e_2,\ldots,e_k\texttt{)}), \\
\qquad \text{if there is a constructor } \texttt{C} \text{ and} \\
\qquad \text{expressions } e_1,\ldots,e_k \text{ such that:} \\
\qquad \Phi_M(\textbf{new } \texttt{C(}e_1,e_2,\ldots,e_k\texttt{)}) = f(t_1,\ldots,t_n)
\end{cases} \\
\Phi_S(t_1 = t_2)) & \triangleq & \Phi_S(t_1) = \Phi_S(t_2), \text{if } t_1 \text{ and } t_2 \text{ are not of model type} \\
 & & \qquad \text{otherwise handled the same way as} \\
 & & \qquad \text{predicate symbols (previous case)} \\
\Phi_S(\neg\phi) & \triangleq & \neg\Phi_S(\phi) \\
\Phi_S(true) & \triangleq & true \\
\Phi_S(false) & \triangleq & false \\
\Phi_S(\phi_1 \circ \phi_2) & \triangleq & \Phi_S(\phi_1) \circ \Phi_S(\phi_2), \text{ if } \circ \in \{\wedge, \vee, \Rightarrow\} \\
\Phi_S(\forall x.\ \phi) & \triangleq & \forall x.\ U_{\widehat{M}}(x) \Rightarrow \Phi_S(\phi) \\
\Phi_S(\exists x.\ \phi) & \triangleq & \exists x.\ U_{\widehat{M}}(x) \wedge \Phi_S(\phi)
\end{array}
$$

where the shaded parts are added only if the quantified variable is of the type to which the model class was mapped.

Figure 9.3: Definition of translation Φ_S

reverses translation Φ_M as expressed by the condition—modulo the application of function $\overline{\gamma}$.

In fact, there might be multiple methods that satisfy the condition—think of method `union` and `bogusUnion`. If so, any of those methods can be selected since their equivalence has to be formally proven, as we will see below.

Note that the translation of operator "=" is different if the operands are of model types and if they are of some other type. In the former case, the definition over function and predicate symbols apply: to which model-class method the operator is mapped depends on the user-specified mapping. In practice, it is typically (but not necessarily) the `equals` method.

If the operands are not of model type, then "=" is translated to "=" (or the equivalent symbol of the target prover). Although this is in line with the definition of function Φ, it might not be the desired translation. In particular, one might want to define equality over the elements of a model

class by the `equals` method of the specific element type at hand, and not by reference equality. This issue is discussed in Chapter 10.

Example 9.7. Consider the translation of term $A\ Un\ \{\}$ to the context of model class `JMLObjectSet`. Both of the two nonlogical symbols of the term, Un and $\{\}$, can be unconditionally mapped to a method of the model class. The former to method `union` and the latter to the parameterless constructor. Thus, the whole term can be translated unconditionally, too. More specifically, we have:

$$\Phi_M(A\texttt{.union(JMLObjectSet())}) \ \equiv\ A\ Un\ \{\}$$

From which we get that:

$$\begin{array}{rcl}\Phi_S(A\ Un\ \{\}) & \equiv & \overline{\gamma}(A\texttt{.union(JMLObjectSet())}) \\ & \equiv & \widehat{union}(A, \widehat{JMLObjectSet}()) \end{array}$$ □

Proof Obligations. The requirement on mappings prescribes that there should be at least one unconditional mapping for each symbol of S. However, it does not rule out methods with mappings that can only be reversed under a certain condition. Therefore, what remains to be shown is that the functionalities of methods that are mapped to the same symbol of S are equivalent provided that the condition (if any) under which their mapping can be reversed holds.

For instance, we need to prove that the functionality of a call to constructor `JMLObjectSet(e)` is equivalent with that of method `insert` provided that the receiver object of the method denotes the empty set.

This kind of proof obligations can be formalized as follows. Assume that for some symbol f, method m fulfills the requirement. Then for each method n that is also mapped to symbol f (even if n also fulfills the requirement), we have to show that the following is a theorem in the context of $\widehat{M}$:

$$\begin{array}{l}\forall\, x_1, \ldots, x_p,\ y_1, \ldots, y_q. \\ \quad \Phi_S(t_m^1 = t_n^1) \wedge \ldots \wedge \Phi_S(t_m^k = t_n^k)\ \Rightarrow\ \widehat{m}(x_1, \ldots, x_p) \stackrel{eq}{=} \widehat{n}(y_1, \ldots, y_q)\end{array} \tag{9.4}$$

where

- the $t_m^i = t_n^i$ equalities are derived by applying translation function Φ_M on methods m and n, and taking pairwise the i-th arguments of the resulting function applications. Formally:

$$\begin{array}{rcl}\Phi_M(x_1\texttt{.}m\texttt{(}x_2, \ldots, x_p\texttt{)}) & = & f(t_m^1, \ldots, t_m^k) \\ \Phi_M(y_1\texttt{.}n\texttt{(}y_2, \ldots, y_q\texttt{)}) & = & f(t_n^1, \ldots, t_n^k)\end{array}$$

- symbol $\stackrel{eq}{=}$ denotes operator "=" if the operands are not of model type, otherwise an application of the hat-function to which symbol "=" is translated by Φ_S (*i.e.*, typically function $\widehat{equals}$).

Example 9.8. According to Figure 8.3 on page 121, there are three symbols of Isabelle's `HOL/Set` theory that are mapped to by multiple methods of model class `JMLObjectSet`: *insert*, "=", and "−". The corresponding proof obligations are the following:

$$\forall\, x_1, x_2, y_1.\ x_2 = y_1 \wedge \widehat{equals}(x_1, \widehat{JMLObjectSet}()) \Rightarrow \widehat{equals}(\widehat{insert}(x_1, x_2), \widehat{JMLObjectSet}(y_1))$$

$$\forall\, x_1, x_2, y_1.\ \widehat{equals}(x_1, y_1) \wedge \widehat{equals}(x_2, \widehat{JMLObjectSet}()) \Rightarrow \widehat{equals}(x_1, x_2) = \widehat{isEmpty}(y_1)$$

$$\forall\, x_1, x_2, y_1, y_2.\ \widehat{equals}(x_1, y_1) \wedge \widehat{equals}(x_2, \widehat{insert}(\widehat{JMLObjectSet}(), y_2)) \Rightarrow \widehat{equals}(\widehat{difference}(x_1, x_2), \widehat{remove}(y_1, y_2))$$

These proof obligations have to be shown in the theory that is derived from the specification of the model class. The derivation procedure is presented in the next section.

We show in detail how the first proof obligation is derived. Both method `insert` and constructor `JMLObjectSet(e)` are mapped to symbol *insert*. The premise of the proof obligation is derived from the following equalities:[4]

$$\begin{array}{lcl} \Phi_M(x_1\texttt{.insert(}x_2\texttt{)}) & = & insert\ x_2\ x_1 \\ \Phi_M(\texttt{new JMLObjectSet(}y_1\texttt{)}) & = & insert\ y_1\ \{\} \end{array}$$

This leads to the premise:

$$\Phi_S(x_2 = y_1) \wedge \Phi_S(x_1 = \{\}) \ \equiv\ x_2 = y_1 \wedge \widehat{equals}(x_1, \widehat{JMLObjectSet}())$$

Note that the "=" symbol is mapped to "=" in the first conjunct because the arguments are not of model type. In the second conjunct the symbol is mapped to an application of symbol $\widehat{equals}$ because the arguments are of model type.

The consequence of the proof obligation consists of an application of function symbol $\widehat{equals}$ because the method and the constructor yield values of a model type. The arguments of the function application correspond to the encoding of the method call x_1`.insert(`x_2`)` and the constructor call `new JMLObjectSet(`y_1`)`: $\widehat{insert}(x_1, x_2)$ and $\widehat{JMLObjectSet}(y_1)$, respectively. □

Revisiting Example 9.5. Recall that Example 9.5 highlighted the necessity of proving that method `bogusUnion` possessed the properties of symbol

[4] Note that symbol *insert* fulfills requirement (9.3) as it is mapped to by method `insert` unconditionally.

Un, the symbol to which the method is mapped. Our approach enforces this by posing the following proof obligation:

$$\forall\, x_1, x_2,\ y_1, y_2.\ \widehat{equals}(x_1, y_1) \wedge \widehat{equals}(x_2, y_2) \ \Rightarrow \\ \widehat{equals}(\widehat{union}(x_1, x_2), \widehat{bogusUnion}(y_1, y_2))$$

That is, essentially we need to prove that method `union` and `bogusUnion` are equivalent. Given the weak specification of method `bogusUnion`, this cannot be proven. □

9.3.3 Proving that Φ_S is a Standard Interpretation

It remains to prove that Φ_S is a standard interpretation. The procedure is the same as for translation Φ_M: we have to show that the three sufficient obligations hold for the standard translation Φ_S.

Model Theory. As a first step, the context and theory in which the obligations are to be proven needs to be constructed. As noted above, the context is denoted by $\widehat{M}$, and the theory is formed by the axiom system that is extracted from the specifications of model class M. In the sequel, we will call this theory the *model theory* and assume that method signatures in M only refer to the enclosing type and type `Object`. In practice, this is typically the case for methods that correspond to the operations of the mathematical structure that M represents.

The model theory is obtained in three simple steps for a model class M:

1. Two new types are declared: *Object* and M.
2. Each method m of M is turned into a function symbol $\widehat{m}$ and its signature is declared based on m's signature using the two newly declared types.
3. Each invariant and method specification of M is turned into an axiom. For an invariant Inv, the resulting axiom is:

   $\forall\ this.\ \overline{\gamma}(Inv)$

   For the specification of method m with one explicit parameter p, precondition P, and postcondition Q, the resulting axiom is:[5]

   $\forall\ this, p.\ \overline{\gamma}(P$ `==>` $Q[$`this.`$m($`p`$)/\$`result`$])$

Example 9.9. Consider method `isProperSubset` of class `JMLObjectSet`. In the first step, types *Object* and *JMLObjectSet* are declared. Next, the signature of $\widehat{isProperSubset}$ is declared. The only explicit parameter of the method is of type `JMLObjectSet`, and the method yields a boolean value. Thus, the signature to generate is the following:

[5] For the specification of constructor C, the substitution to perform on Q is $C(p)$/`this`.

$$\widehat{isProperSubset} : \ JMLObjectSet \ \times \ JMLObjectSet \Rightarrow bool$$

In the third step, axioms over `isProperSubset` are stated. For instance, the equational theory (*i.e.*, the specification of method `equational_theory`) of `JMLObjectSet` contains the following equation:

```
s.isProperSubset(s2) == (s.isSubset(s2) && !s.equals(s2))
```

The equation is part of the specification of method `equational_theory`, which has 4 parameters: s, $s2$, $e1$, and $e2$ (the method has no implicit parameter as it is static). Thus, in the third step, the equation is turned into an axiom that quantifies over these variables. The "body" of the axiom is yielded by the application of function $\overline{\gamma}$ on the specification: (1) the calls on methods `isProperSubset`, `isSubset`, and `equals` are encoded by the function applications $\widehat{isProperSubset}(s, s2)$, $\widehat{isSubset}(s, s2)$, and $\widehat{equals}(s, s2)$, respectively; and (2) the logical operators are encoded. This results in the following axiom of the model theory:

$$\begin{array}{l} \forall \ s, s2, e1, e2. \\ \qquad \widehat{isProperSubset}(s, s2) = (\widehat{isSubset}(s, s2) \wedge \neg \widehat{equals}(s, s2)) \end{array} \tag{9.5}$$

□

Remark. As mentioned earlier, declaring and using type *Object* in function signatures does not allow one to handle, for instance, sets of sets. A more flexible solution, provided the target theorem prover supports type variables, is to declare M as a polymorphic type (*e.g.*, "α *JMLObjectSet*" in Isabelle). Thereby, the declaration of type *Object* is not needed and in signatures of function symbols it is to be replaced by α.

Once the model theory is created, we have to show that the formulas that correspond to the three obligations for translation Φ_S are theorems of the model theory.

Axiom Obligation. To prove the axiom obligation, we have to show that every axiom and definition ϕ of S is a theorem of $\widehat{M}$. That is, formula $\Phi_S(\phi)$ is a theorem of the model theory.

Example 9.10. Consider definition `psubset_def` of the `HOL/Set` theory:[6]

$$\forall \, A, B. \ A < B \ = \ A <= B \ \wedge \ \neg \, A{=}B$$

[6]The formula is equivalent with definition `psubset_def` on page 124, only the syntax is adapted and the binding of the implicitly bound variables is made explicit.

Applying Φ_S on the definition, we get the following formula to prove:

$$\forall\, A, B.\ U_{\widehat{M}}(A) \Rightarrow U_{\widehat{M}}(B) \Rightarrow \widehat{isProperSubset}(A, B) = (\widehat{isSubset}(A, B) \wedge \neg\, \widehat{equals}(A, B))$$

The formula can be trivially proven by axiom (9.5). □

Implicit Axiom Obligations. It is important to note that not only the axioms and definitions of S have to be translated, but also those rules of the target theorem-prover that are applicable to instances of S and, thereby, express properties of structure S. For instance, Isabelle's higher-order logic is based on a few axioms that (among others) express reflexivity of equality and Leibniz-equality [107]. Although these axioms are not specific to a given structure S, they express properties of the symbols of every structure S. Therefore, these two axioms also have to be translated to and proved by the model theory to make sure that the corresponding properties hold in the model class, too.

For model class `JMLObjectSet`, reflexivity of equality yields the axiom obligation $\forall\, t.\ \widehat{equals}(t, t)$, which can be trivially proven by the specification of method `equals`.

The difficulty with the translation of Leibniz-equality is that it is higher order: the property is defined for *every* function symbol or predicate f. Therefore, we need to "approximate" the axiom by posing a separate proof obligation for each function and predicate symbol of S. Then, the resulting formulas can be translated to and proven by the model theory.

For instance, the formula to prove for symbol "<=" in Isabelle's `HOL/Set` theory is the following:

$$\forall\, x_1, x_2, y_1, y_2.\ \widehat{equals}(x_1, y_1) \wedge \widehat{equals}(x_2, y_2) \ \Rightarrow\ \widehat{isSubset}(x_1, x_2) = \widehat{isSubset}(y_1, y_2)$$

Given proper model-class specifications, this kind of proof obligation is trivially provable.

Universe Nonemptiness Obligation. The obligation requires one to prove that universe $U_{\widehat{M}}$ is nonempty: $\exists\, x.\ U_{\widehat{M}}(x)$. As for the consistency proof, the obligation is typically trivially provable.

Example 9.11. The universe predicate to use for class `SmallSet` during the completeness proof can be derived from (9.1) on page 132 by translating it with Φ_S. The resulting predicate is:

$$U_{\widehat{M}}(x) \ \triangleq\ \widehat{equals}(x, \widehat{JMLObjectSet}()) \vee (\exists\, e.\ \widehat{equals}(x, \widehat{insert}(\widehat{JMLObjectSet}(), e)))$$

To prove the obligation, one can pick x to be $\widehat{JMLObjectSet}()$. □

Function Symbol Obligation. The obligation is analogous to obligation (9.2) on page 136 for the consistency proof. Predicate P corresponds to the domain restriction of the function at hand, provided it is partial.

9.4 The Last Hurdle

Once the faithfulness of a mapping has been proven, the encoding of client specifications can make use of it by translating calls to model-class methods according to the specified `mapped_to` clauses. Therefore, the encoding function γ introduced in Chapter 4 has to be adapted accordingly.

For the sound use of specified mappings, there is a last hurdle to overcome: we need to make sure that model fields in client specifications take values from the specified universe. Therefore, we pose the following proof obligation on a model field m of type M declared in type T:

$$\forall\, o, OS.\ allocT(o, OS, T) \Rightarrow U_{\widehat{M}}(OS(o.m)) \tag{9.6}$$

For the proof, one might use the represents clause of the field, which is axiomatized as follows. For simplicity, we assume that represents clauses are of the form:

represents m `<-` `(` P `?` E_1 `:` E_2 `)`

where P is a non-recursive boolean expression (*i.e.*, it must not mention model field m), and E_1 and E_2 yield values of m's type M. The clause is handled as if it was the declaration and specification of a pure method m:

```
pure M m()
  ensures P  ==> \result.equals(E1) &&
          !P ==> \result.equals(E2);
```

The well-formedness checking and axiomatization of the specification is done the same way as described in Chapter 6 for pure methods, except that aliveness of the model value is not stated in the resulting axiom. Note that the value of a model field may depend on the value of other model fields and on the result of pure-method calls. That is, there might be dependencies between model fields and between pure methods and model fields. The handling of such dependencies is also analogous to the handling of dependencies between pure methods.

Example 9.12. Consider the class in Figure 9.4, which represents a node of a linked list with a `value` field that references an object. Model field `_set` is specified to be the set that contains all objects that are referenced by the `value` field of one of the nodes that is reachable from **`this`** through the `next` field.

We omit proving the well-formedness of the represents clause as well as the specifications that are needed for the proof. The only non-trivial

```
class Node {
    nullable Node next;
    Object value;
    model JMLObjectSet _set;
    represents _set <- (next == null ? new JMLObjectSet(value) :
                                       next._set.insert(value));
    // other specifications and methods omitted
}
```

Figure 9.4: Recursive represents clause of a model field

property to prove is well-foundedness, which can be shown by introducing a ghost field to class `Node` that indicates the length of the list starting from the node.

Let us see how formula (9.6) can be proven for field `_set` of class `Node`. Consider the universe predicate of model class `JMLObjectSet` to be composed by its two constructors and method `insert`:

$$\begin{aligned} U_{\widehat{M}}(x) \triangleq\ & \widehat{equals}(x, \widehat{JMLObjectSet}()) \ \vee \\ & (\exists\, e.\ \widehat{equals}(x, \widehat{JMLObjectSet}(e))) \ \vee \\ & (\exists\, s, e.\ U_{\widehat{M}}(s) \wedge \widehat{equals}(x, \widehat{insert}(s, e))) \end{aligned}$$

The axiom that is extracted from the represents clause looks as follows:

$$\begin{aligned} & \forall\, this, OS.\ allocT(this, OS, \texttt{Node}) \ \Rightarrow \\ & (OS(this.next) = null \ \Rightarrow \\ & \widehat{equals}(OS(this.\_set), \widehat{JMLObjectSet}(OS(this.value)))) \wedge \\ & (OS(this.next) \neq null \ \Rightarrow \\ & \widehat{equals}(OS(this.\_set), \widehat{insert}(OS(OS(this.next).\_set), OS(this.value)))) \end{aligned}$$

Since the represents clause of `_set` is defined recursively, we need to prove formula (9.6) by induction, more specifically, by induction over the length of the list.

In the base case, the length is 1. Consequently, **`this.next`** is **`null`**, which allows us to deduce the formula on the third line of the axiom. Since the formula is an instance of the second disjunct of $U_{\widehat{M}}(OS(this.\_set))$, the base case is proven.

In the step case, the length is greater than 1. Consequently, **`this.next`** is non-null, which allows us to deduce the formula on the last line of the axiom. The formula is an instance of the second conjunct in the third disjunct of $U_{\widehat{M}}(OS(this.\_set))$, leaving us with formula $U_{\widehat{M}}(OS(OS(this.next).\_set))$ to prove. This follows from the induction hypothesis, which can be applied because the length of the list starting from the node denoted by **`this.next`** is smaller than that of **`this`**. □

Chapter 10

Discussion

This chapter discusses various aspects of our approach to the faithful mapping of model classes, in particular, cases when the faithfulness of a mapping cannot be proven, the potential for automating certain parts of faithfulness proofs, a subsidiary application of our approach, issues with the handling of equality, and guidelines for writing model classes.

10.1 Mismatches Between Model Classes and Mathematical Structures

The interface of a model class is typically developed independently from the structures that it may potentially be mapped to. This is justified by the fact that model classes are used by different program verifiers equipped with different back-end theorem provers. Different theorem provers provide different mathematical structures and functions; thus, typically it is not possible to develop model classes that can be faithfully mapped to all seemingly suitable structures. Consequently, there might be a mismatch between the model class and the theorem prover's structure.

In the previous chapter, we assumed that every model method can be mapped to some symbol of the target context, and that every symbol can be mapped back unconditionally to some method. We have seen that in this case a standard translation can be derived, and the theory interpretation can be performed both for the consistency and for the completeness proof.

According to our experience, often there is no such perfect match between methods and symbols, and no mapping can be given for a model method or a symbol of the structure. In such cases, there is a mismatch that cannot be bridged: equivalence of the theories of M and S cannot be shown. However, the "direction" of the mismatch makes a difference in the consequences.

If a method of M cannot be mapped to S then we cannot be sure if JML^{--} specifications that contain calls to the method are consistent, and

if the method semantically corresponds to some mathematical operation. In such situations, one needs to pick a richer target structure where the mapping is possible.

The situation is better if a symbol of S cannot be translated to M. Although the theory of S cannot be fully interpreted in the theory of M, the theory of M can still be fully interpreted in the theory of S. Thus, the correspondence can still be shown between all methods that are defined by the interface of M and the symbols of S to which the methods are mapped to. We call this situation *observational faithfulness*, which seems to be sufficient for the use of `mapped_to` clauses for the verification of client code, because clients of a model class can only call methods that are available in the model class, and the consistency and completeness of those methods are still shown.

However, the following example demonstrates that observationally faithful mappings should only be used under certain conditions.

Example 10.1. Assume class `C` and theory T are specified as follows.[1]

```
mapped_to("T");
class C {
    mapped_to("T","f(p)");
    static int m(int p)
      ensures \result >= 10;
    { ... }
}
```

theory T:

axiom: $\forall x.\ f(x) \geq 10$
axiom: $\forall x.\ f(x) = g(x)$
axiom: $\forall x.\ g(x) = 42$

Observational faithfulness of class `C` and theory T can be trivially proven since (1) the postcondition and the first specification of T are equivalent and (2) the other two specifications cannot be translated to the language of `C` since symbol g is not mapped to by methods of `C`.

Now, assume a client code contained the specification `m(a) == 42` for some integer `a`. Although based on the specification of the method we cannot deduce whether the expression holds or not, the translation of the formula, $f(a) = 42$, is trivially provable in theory T. That is, we can prove more properties about `m` after the translation than before. □

To prevent such situations, we pose the following requirement on observationally faithful mappings:

> Theory T_S of structure S must be a conservative extension of theory T_S^M that we get from T_S by removing all symbols that are not mapped to by M together with all axioms that contain such symbols. (10.1)

In the previous example, T_S would consist of the three axioms above, while T_S^M would consist of the first axiom only. As we can see, T_S is not a

[1] The `mapped_to` clauses only contain information that is necessary for the example.

conservative extension of T_S^M, because $\forall x.\ f(x) = 42$ is a formula but not a theorem of T_S^M, while it is a theorem of T_S.

Note that if only observational faithfulness can be shown between M and S, then requirement (9.3) on page 140 obviously does not hold, because there are symbols f for which the prescribed condition does not hold. Therefore, a more liberal requirement is needed:

> For each n-ary function and predicate symbol f of S *that is mapped to by a method or constructor of* M, and for variables $x_1, \ldots, x_n$ there is at least one method m or constructor C of M, and expressions $e_1, \ldots, e_k$ with free variables $x_1, \ldots, x_n$ such that either $\Phi_M(e_1.m(e_2, \ldots, e_k)) = f(x_1, \ldots, x_n)$ or $\Phi_M(C(e_1, e_2, \ldots, e_k)) = f(x_1, \ldots, x_n)$ holds. (10.2)

When proving observational faithfulness of a mapping, all proof obligations remain the same as presented in the previous chapter. In particular, due to requirement (10.1), one can use the specified target theory when performing the consistency proof (*i.e.*, T_S instead of T_S^M).

Observational faithfulness is important in practice as typically there are many operations in S that cannot be mapped to M. For instance, a theorem prover with a higher-order logic typically supports operations like filter or map, which are not expressible in JML^{--}, which is first order.

10.2 Automation of Faithfulness Proofs

In the previous chapter, we have seen how proof obligations and the context in which they have to be proven are extracted from the specifications of M and S. At the time of writing, our approach does not have tool support and proof scripts have to be written manually. In this section, we briefly discuss which parts of our technique could be automated.

Consistency Proof. As Example 9.3 on page 135 suggests, proving consistency may be fully automated. (1) The translation of model specifications is performed by the application of translation function Φ_M, which mainly performs syntactic substitutions based on `mapped_to` clauses. (2) The lemma was proved without any user interaction using a powerful tactic of Isabelle.

For more complicated properties the mere use of tactics might not suffice and one might need to help the prover by giving hints at which theorems of S to use. Still, the basic proof skeleton can be generated.

As mentioned before, if the universe predicate is defined inductively, the lemmas that correspond to the axiom and the function symbol obligations

have to be proven by structural induction. In such cases, even the generation of the proper lemmas is non-trivial, let alone the generation of proof scripts. The same holds for the completeness proof, too. However, apart from artificial examples, we have not yet seen the need for an inductive universe predicate.

Completeness Proof. As shown in Section 9.3.3, the first step of the completeness proof is the generation of the model theory, which is extracted from model specifications using the $\overline{\gamma}$ function. As Example 9.9 on page 144 suggests, this step can be fully automated.

The next step of the completeness proof is the generation of lemmas. To do so, one needs to find the proper reverse translation of symbols, that is, for each symbol a method and a number of expressions have to be found that satisfy requirement (9.3). In general, this is difficult to fully automate because finding the proper expressions for $e_1, \ldots, e_k$ is non-trivial. However, in many cases the task is rather straightforward since the expressions often correspond to single variables. For instance, when translating the general term *A Un B* for set union, the expressions that have to be found are A and B. Therefore, it seems that large portions of the reverse translation can be automated.

Once lemmas have been generated, they have to be proven. This step is also less trivial than for the consistency proof. First, even the application of automated tactics typically requires one to manually select the set of axioms to be used for proving a given lemma, because selecting all axioms (or applying heuristics that select all axioms that mention symbols that appear in the lemma) might cause the tactic to loop. Second, beyond trivial cases, tactics often fail to find the proper instantiations of extracted axioms. In such cases, further hints need to be given to the prover, for instance, by the insertion of simple helper lemmas. Third, the specification of the model class may be too weak to verify some axioms or definitions of the structure. If so, the missing specifications need to be identified, which requires manual intervention.

Thus, it seems that the automation of the completeness proof can only be partial and manual intervention is needed. However, the effort is justified by the increased quality of the model class specification.

Applying Mappings. Once consistency and completeness have been proven, calls to model-class methods that occur in client specifications can be directly mapped to terms of the target theory. The mapping is based on specified `mapped_to` clauses and can be performed automatically by a verification tool.

Remark. Our experience regarding automation is based on using Isabelle and its `HOL/Set` theory. However, we expect similar results with other theo-

rem provers and theories that provide built-in tactics: (1) the translation of formulas is mainly a syntactic operation, thus its automation is independent of the underlying prover and logic; (2) the automation in the proof effort is mainly based on the tactics of the underlying theorem prover.

10.3 Discovering and Checking Redundancy

An interesting side-effect of proving completeness of model-class specifications is that redundant methods and specifications can be discovered in the model class, and one can check if specification elements marked as redundant are indeed implied by other specification elements.

- Proof obligation (9.4) on page 142 allows one to prove that certain methods are "redundant" in the model class, that is, expressible by other methods. Although such methods and their specifications could be removed without altering the capabilities of the model class, they are part of model-class interfaces in order to make specifications more comprehensible.

- A specification element is redundant if (1) the axiom that is extracted from it is never used in the completeness proof and (2) the specification element does not mention "redundant" methods (see previous point). To illustrate the intuition behind (2), consider the following equation of the equational theory:

  ```
  new JMLObjectSet(e1).equals(new JMLObjectSet().insert(e1))
  ```

  The axiom extracted from the equation is not needed in the completeness proof, because there is no function symbol that corresponds to the "redundant" one-argument constructor. However, the specification is not redundant because it defines the "redundant" constructor, which is part of the model-class interface.

- To check if specification elements marked as redundant are indeed redundant, we just have to state the extracted formulas of such specifications as lemmas (and not axioms) when proving completeness of the model class. Lemmas that are provable using the axioms (extracted from specification elements not marked as redundant) confirm that the corresponding specification elements are indeed redundant.

  As we will see in the next chapter, this approach allows us to compare the strength of the equational theory and the method specifications of a model class.

The identification and checking of redundant specifications further improve the quality of model-class specifications.

10.4 Equality of Model Classes and Their Elements

An important issue of the mapping of model classes is the handling of equality. In particular, the handling of equality over model-class *instances* has to be distinguished from the handling of equality over the *elements* of collections that model classes represent.

Model-Class Instances. Since model-class instances are treated as mathematical objects, their equality is always determined by the `equals` method of the class. Therefore, it is incorrect to apply operator "`==`" on instances of model classes. In such cases, we replace it by a call to `equals`.

The `equals` method of a given model class may have a `mapped_to` clause just like any other method of the class. Typically, but not necessarily, the clause prescribes a mapping to the "=" operator (or equivalent) of the mathematical structure, which is "overloaded" in the target context to mean equality of the given structure. For instance, in Isabelle, the "=" operator has the following meaning when applied over instances of α *set*:

$$\forall A, B.\ (A = B) = (\forall x.\ (x : A) = (x : B)) \tag{10.3}$$

During the consistency and completeness proofs, one has to show that this meaning of equality is indeed in line with the semantics of the `equals` method of the model class.

Elements of an Instance. The elements of model-class instances are typically objects, therefore, the question arises whether their equality should be determined by reference equality or by the application of the `equals` method of the corresponding type.

There is no good answer to this question, because the desired meaning of equality depends on the client code one wants to specify. Therefore, the JML model library provides multiple model classes for the same mathematical structure, and the model classes only differ in the handling of equality over the elements. For instance, besides class `JMLObjectSet`, the library contains class `JMLValueSet`, which considers elements to be values. That is, the elements are still objects, but their equality is determined by the `equals` method of the element type.

If the equality of elements is reference equality, the translation is usually straightforward because Java's "`==`" operator corresponds to Leibniz-equality [29]. Thus, the operator can be translated to the "=" symbol (or equivalent) of the target theory.

However, if the equality of elements is deep equality (*i.e.*, with the `equals` method), then this translation does not work. The main issue is that the target theory typically contains definitions and lemmas that use the "=" symbol over elements of the structure. For instance, lemma `singletonD` in

theory `HOL/Set` states that for every element `a` and `b` the following holds: `b : {a} ==> b = a`. In order to carry over the "value semantics" of the elements of class `JMLValueSet`, the semantics of the "=" symbol would need to be re-defined according to the semantics of the `equals` method of the corresponding element type.

This seems to be feasible at first sight: as seen above, the meaning of the symbol could be "overloaded" for type $\alpha\ set$. However, the situation is different for objects that are not purely mathematical, but are objects with state: The results of `equals` methods (typically) depend on the heap, thus their semantics cannot be expressed by the "=" operator, which takes no argument for the heap.

Therefore, we follow the approach of JML and use a "copy" of a given theory in which all usages of the "=" operator applied on elements are replaced by applications of the uninterpreted function symbol $\widehat{equals}$, which encodes the specification of the corresponding `equals` method.

Note that the resulting theory remains consistent provided that the original theory was consistent and the `equals` method defines an equivalence relation (as it is supposed to).

10.5 Guidelines for Writing Model Classes

We close this chapter on discussion by providing a number of guidelines for the writing of model classes. The guidelines apply to the methods, universes, and specifications of model classes. Based on our experience, we believe that (observational) faithfulness proofs get considerably simpler if these guidelines are taken into account.

Methods. The main idea behind model classes is that they represent well-known mathematical structures. This should also apply to the methods of a model class: one should refrain from adding "exotic" features and, instead, should focus on the elementary ones.

Providing a rich interface for a model class may seem appealing, because it helps to specify client code in a simpler and more compact way. However, there are a few points that speak against large model-interfaces. First, it is difficult to predict what "non-standard" operations would actually be useful to include in the interface, because for different specific domains different operations are useful.

Second, finding a mathematical theory to which the class can be mapped becomes more difficult, or even impossible. Recall from Section 10.1 that if a method cannot be mapped, then no guarantee can be given on the consistency and semantic meaning of specifications that use that method. Therefore, one should be careful when using such methods in client specifications, if at all.

Note also that requirement (10.2) on page 151 does not hold if a model class provides an operation that is just a special case of a more fundamental operation, which is not provided by the class. For instance, the requirement would not hold if model class `JMLObjectSet` provided method `remove` but not method `difference`. Thus, when designing a model class, one should first think of basic operations of the structure represented by the class.

Universes. The above consideration applies for universes as for methods: model classes are meant to represent standard structures, thus one should aim for constructing model classes with "standard" universes. That is, for instance, a model class that represents mathematical sets should not provide a universe with instances representing only the empty set and the singleton set, like `SmallSet` in Figure 9.1 on page 131. Instead, the class should represent finite or infinite sets for which it is more likely that a matching theory exist in state-of-the-art theorem provers.

Recall that two of the three sufficient obligations of standard interpretations trivially hold if the source and the target universes match. This is a particularly large gain for the function symbol obligation, which otherwise requires numerous non-trivial properties to be proven.

The remaining obligation, the axiom obligation also becomes simpler to prove, because one does not need to make use of the universe predicate.

Specifications. In the course of our case study on model class `JMLObjectSet`, we noticed that the class contains specification elements that express certain *properties* of a given operation (typically relating the operation to another one), and not its *definition*.[2] Such specifications make both the consistency and the completeness proof more tedious. During the consistency proof, multiple specifications have to be translated and proved for a given operation, instead of just one or a few. And, if an operation is specified through several specification elements, then it is more likely that eventually one ends up with an incomplete specification for the operation.

Therefore, writing specifications that correspond to definitions is preferable. Other specifications that describe certain relations of model methods (and might be helpful for clients of the model class) should be marked as redundant. This is facilitated by specific constructs in JML.

[2] A concrete example will be given in the next chapter.

Chapter 11

Case Study: Mapping `JMLObjectSet` to `HOL/Set`

In this chapter, we demonstrate the proposed technique through a case study: we map model class `JMLObjectSet` to Isabelle's `HOL/Set` theory, and prove that the mapping is faithful. Overall, we considered 17 operations of the model class: 2 constructors, 9 query methods, and 6 methods that yield `JMLObjectSet` instances. These were all the constructors and methods that remained after the simplification step that we discuss below.

All proofs were carried out in Isabelle. The proof scripts contained a total of ca. 420 lines of code (LOC) without comments and empty lines. Consistency of the mapping was proven in ca. 100, completeness in ca. 155 LOC. The model theory of the class consisted of ca. 75 LOC. Equivalence of the equational theory and the method specifications (described in Section 11.2) was proven in ca. 90 LOC. As mentioned above, our technique does not yet have tool support, thus all steps of the case study were performed manually.

All proofs of the case study and the full mapping of the class are available online [136].

11.1 Simplification and Division of Specifications

Since we were interested in the mapping of `JMLObjectSet` and its methods to an Isabelle theory, we first removed all methods that provided object-oriented features irrelevant for the mapping of the model class. These methods included, for instance, `clone`, `hashCode`, and `toString`. In our opinion, such methods need not be part of model classes if one thinks of them as mathematical structures. As mentioned already, conceptually we do not consider model classes to inherit from class `Object`.

Next, we removed all implementation details, including non-public methods and specifications, since our approach is merely based on the public specification of model classes. Additionally, we removed public methods

that only provide syntactic sugar, for instance, the constant field `EMPTY` that represents the empty set and is equivalent with the result of constructor `JMLObjectSet()`. As mentioned in Section 9.2, only method specifications that describe normal behavior need to be treated by our approach. Thus, we removed all other method specification cases.

As mentioned before, we do not consider the handling of ghost fields. Thus, in the case study we removed them from the model class together with all specification expressions that referred to them. The removal of the ghost fields did not change the important characteristics of the model class since they were not used in a way that would have altered the underlying semantics of the class.

The class has four ghost fields, two of which are inherited from `Object` and are not related to the model class in any way. The other two ghost fields are `containsNull` and `elementType`. The former has value true if and only if the represented set contains the **`null`** value. After removing the ghost field, this information can be queried by the call **`this`**`.has(`**`null`**`)`.

The ghost field `elementType` is of type `\TYPE`, which is a type introduced by JML and represents the kind of all Java types [77]. The field gives an upper bound on the types of the set-elements, that is, all elements must be a subtype of `elementType`. By dropping the field, the information about this upper bound is lost. However, this does not change the main characteristics of the model class that we are interested in.

To focus on the main ideas of our approach, we decided not to handle methods that referred to non-primitive types other than `Object` and `JMLObjectSet`. For instance, methods that convert `JMLObjectSet`s to other model and non-model types, such as `Object[]`. The handling of these kinds of methods is possible once one has provided mappings for all types that are mentioned in their signatures.

What remained was 17 methods with method specifications describing normal behavior, and a large equational theory describing the relations between the methods.

Division of Specifications. We analyzed the specification of `JMLObjectSet` and found that method specifications and the equational theory were highly redundant. We illustrate this redundancy by method `union`. The equation defining the method in the equational theory and the specification of the method is given in Figure 8.2 and 8.3 on pages 120–121, respectively. It is easy to see that, after proper substitutions, the two specifications express the same property.

Thus, we decided to split specifications into two parts: one containing only the equational theory and the other containing only the method specifications. This allowed us to analyze their relation, discussed in Section 11.2.

Remark. It is not always the case that the equational theory of a model class and its method specifications are redundant. For instance, model classes `JMLObjectToObjectRelation` and `JMLValueValuePair` specify the behavior of the class in great majority by method specifications. One could as well specify a model class mainly or entirely by an equational theory. Thus, in general, faithfulness should be proven using both the equational theory and the method specifications together.

11.2 Proving Faithfulness of Mapping

Specifying the Mapping. The next step was to specify the mapping of the model class and its methods. The resulting mapping of the methods that we consider in this chapter was shown in Figure 8.3 on page 121.

The mapping of the methods was mostly straightforward, we briefly mention three non-trivial cases. Method `choose` yields an arbitrary element of the set in case it is not empty. Although Isabelle's set theory has no equivalent function, the method directly corresponds to Hilbert's ϵ-operator, written as "$SOME\, x.\ P(x)$" in Isabelle, denoting some x for which $P(x)$ is true, provided one exists [106]. In our case, P simply needs to express set membership of x as follows: $\nu(\texttt{choose}) \equiv \lambda\{this.\ SOME\ x.\ x : this\}$.

Another non-trivial case was the mapping of method `remove`. As discussed in Chapter 9, theory `HOL/Set` does not contain a corresponding function, thus the method is mapped to term $this - (insert\ elem\ \{\})$, which contains 3 function symbols of the target theory.

The handling of equality was rather straightforward. As discussed in Section 10.4, equality of model-class instances is determined by the `equals` method of the class. The method is mapped to the "=" operator, which expresses set equality when applied over instances of $\alpha\ set$ (see formula (10.3) on page 154). Since `JMLObjectSet` represents a set of *objects*, equality over the elements of sets means reference equality. Since Isabelle's "=" operator on some element of type α corresponds to reference equality of objects, we simply map Java's `==` operator to operator "=" when applied to elements stored in a `JMLObjectSet`.

The Universe Predicate. Although `HOL/Set` is a theory of *infinite* sets and model class `JMLObjectSet` represents a *finite* set of objects, the universe predicate can be defined to be the trivial one, because faithfulness[1] of the mapping can be proven without the relativization of quantifiers. This is due to the fact that both the model class and the data type is equipped with set comprehension, and that the model class corresponds to a finite set only because JML syntactically restricts the predicate of set comprehensions to

[1] In fact, only *observational* faithfulness can be proven, as discussed below.

describe a finite set of objects (see page 120). However, this restriction does not effect the properties of the usual set operations that the class provides.

Consistency Proof. In the next step, we proved consistency of the model class. That is, we proved that the translation defined by the specified mappings yields a standard interpretation to Isabelle's `HOL/Set` theory.

We found one inconsistent equation in the equational theory. This equation intended to describe a relation between method `remove` and `insert` as follows:

```
s.insert(e1).remove(e2).
  equals(e1 == e2 ? s : s.remove(e2).insert(e1))
```

where `s` is a `JMLObjectSet` instance, and `e1` and `e2` are two objects. The specification expresses that if `e1` and `e2` refer to the same object then inserting and removing the object into and from set `s` yields a set equal to `s`; otherwise, the order of performing the two operations is interchangeable.

Although this might look correct at first sight, the attempt to formally prove its correctness reveals that it is incorrect in case `s` contains `e2`, and `e1` and `e2` refer to the same object. In this case, the insertion yields some set s' that contains the same objects as `s` and the remove operation yields some set s'' that contains the same objects as s' except the object referenced by `e2` (and `e1`). Thus, this set cannot be equal to `s`.

This problem was directly pointed out by Isabelle via the open goal that remained after applying the automatic tactic *auto* on the corresponding lemma. The open goal was: $e2 : s \Rightarrow False$, expressing that the property does not hold in case `s` contains `e2`.

The buggy equation could be easily fixed after the problem was caught and all specifications of the equational theory and the method specifications could be proven trivially using the *auto* tactic of Isabelle. As a consequence, we proved that the (corrected) specification of the model class is consistent.

In general, the axiomatic JML specifications seem to be more error-prone than the conservative Isabelle specifications. Therefore, we expect other model classes to contain similar bugs, which our technique can reveal.

Equational Theory vs. Method Specifications. While it was easy to notice the large overlap of properties specified by the equational theory and the method specifications, it was not trivial to see whether they are equivalent. Thus, after having proved that the specifications are consistent, we proved their equivalence formally using Isabelle.

The procedure of proving the equivalence was the following. First, we declared the signatures of symbols in the model theory the same way as described in Section 9.3.3. Then, when proving that the equational theory implies the method specifications, we stated axioms based on the equational theory and generated lemmas based on the method specifications. Finally,

we attempted to prove the lemmas using the axioms. The other direction was proved analogously.

We found that the equational theory and the method specifications were not equivalent, and none of them contained stronger specifications than the other. That is, while proving either direction, some lemmas could not be proven without strengthening some of the axioms or adding new ones. Four additional equations had to be added to the equational theory and one postcondition had to be strengthened in the method specifications in order to prove their equivalence. We give an example for both directions.

Example 11.1. The equational theory defines method `isEmpty` in terms of method `int_size`, which is excluded from our case study (see Section 11.3). Therefore, we relied on the two specifications that are marked as redundant in the equational theory and that mention method `isEmpty`:

```
new JMLObjectSet().isEmpty()   and   !s.insert(e1).isEmpty()
```

These express that a newly-allocated set is empty and that a set into which an element is inserted is not empty. These specifications do not imply the property stated in the postcondition of method `isEmpty`:[2]

```
\result == (\forall Object e. !this.has(e))
```

That is, `isEmpty` returns true if and only if the set does not contain any object. The postcondition could not be proven using the two equations because those just express *properties* of `isEmpty` (after construction and insertion) while the postcondition gives the *definition* of `isEmpty`. Adding this definition to the equational theory (and thus to the set of axioms used in the proofs) trivially solved the problem. □

Remark. We could not derive the postcondition of method `isEmpty` from the equational theory even if the equations over method `int_size` were taken into account. Therefore, we believe that the definition has to be added to the equational theory in order to make the theory equivalent with the method specifications.

Example 11.2. The specification of constructor `JMLObjectSet(e)` had to be strengthened because the original postcondition `this.has(e)` was not sufficient to prove two specifications from the equational theory, for instance, the equation that relates the two constructors of the class:

```
new JMLObjectSet(e1).
  equals(new JMLObjectSet().insert(e1))
```

The weakness of the constructor's postcondition was again revealed by the open goal while attempting to prove the above expression, and suggested to strengthen the postcondition to express that object `e` is the one and only object contained by the set after construction:

[2]The original JML syntax of quantification has been adapted to that of JML⁻⁻.

```
(\forall Object e1. this.has(e1) <==> (e == e1))
```

The strengthened postcondition allowed us to prove the two remaining specifications in the equational theory. □

To make sure that the added and strengthened specifications did not introduce unsoundness, we proved their consistency.

The result of having proved the equivalence of the equational theory and the method specifications is that one can use one or the other. For instance, one only needs to consider the method specifications for the faithfulness proof, while the equational theory can be marked as redundant.

Completeness Proof. As the last step, we proved completeness of the model class—using the corrected and strengthened specification of class `JMLObjectSet`. As noted above, it would have sufficed to prove completeness either against the equational theory or against the method specifications. Still, in order to gain more experience with our approach, we carried out the proof against both of them.

There were two non-trivial issues with the translation of Isabelle definitions to the model theory. The main issue was that many of the definitions in Isabelle's `HOL/Set` theory use set comprehension (see Figure 8.4 on page 124). Set comprehension can be expressed in JML only by the "implicit" constructor of `JMLObjectSet`, introduced in Section 8.2.1.

The JML Reference Manual [77] does not give a concrete definition for the semantics of the construct, thus we used the meaning that Isabelle defines via the axioms of theory `HOL/Set`. This (1) ensured that we did not introduce unsoundness (provided that Isabelle's axiom is sound), and (2) gave a connection between mathematical set comprehension and the methods of `JMLObjectSet` since the Isabelle axiom refers to set membership, which corresponds to the `has` method of the model class.

The other issue with the translation concerned method `choose`. Recall that the method is mapped to the Hilbert-operator, with a specific predicate expressing set membership. However, no definition or axiom uses the operator with that particular predicate and, as we have seen in Example 9.5, it might lead to unsoundness if we only proved consistency of the method's specification.

Therefore, we translated Isabelle's introduction rule for the operator $P(x) \Rightarrow P(SOME\ y.\ P(y))$ with the appropriate predicate, resulting in the proof obligation:

$$\forall this, x.\ \widehat{has}(this, x) \Rightarrow \widehat{has}(this, \widehat{choose}(this))$$

After having resolved these two issues, the Isabelle definitions could be easily translated to the model theory of the class.

The corresponding lemmas could be proven both by the corrected and strengthened equational theory and by the strengthened method specifications. This means that `JMLObjectSet`'s specification indeed captures the elementary properties of sets.

Most proofs were trivially discharged by giving hints to Isabelle's *auto* and *simp* tactics which axioms to use. When giving such hints, a first approximation is typically to include all axioms that refer to the function symbols that appear in a given proof obligation. However, this might make the tactics loop, requiring one to remove some of the axioms. To decide which ones to remove, one needs to analyze the proof obligation to see which properties may not be needed to complete the proof.

11.3 Mismatching Methods and Functions

Finally, we mention cases where the model class and the Isabelle structure cannot be related to each other.

As one can see in Figure 8.3 on page 121, there is no `mapped_to` clause attached to method `int_size`, which yields the number of elements the set contains. The method cannot be mapped to any term in the target theory since the theory does not define set cardinality and it cannot be expressed by the combination of other functions either. This is because theory `HOL/Set` represents infinite sets.

As discussed in Section 10.1, this mismatch means that our approach can neither guarantee consistency of specifications that mention the method nor that the semantic meaning of the method is indeed set cardinality. Thus, a better choice would be to map the model class to Isabelle's `HOL/Finite_Set` theory, which provides the corresponding function, *card*.

Not surprisingly, due to the higher-order nature of the theory to which the class was mapped, there were definitions that could not be mapped back to the model class. An example is function *image* which takes a function f and a set A as parameters, and yields the image of A under f. The model class does not provide such functionality and it cannot be expressed by other methods of the class. As discussed in Section 10.1, this means that at most observational faithfulness can be shown between the model class and the structure, which is sufficient for the use of the mapping during verification.

Chapter 12

Handling Inductive Structures

Many theorem provers allow one to introduce data types and sets inductively. For instance, data type *list* is typically introduced inductively by two type constructors: `Nil`, which creates the empty list; and `Cons` e ls, which creates a list by adding element e to list ls. The set of natural numbers *nat* is defined inductively by two introduction rules: 0 belongs to the set; and, if n belongs to the set then `Suc` n belongs to the set, too.

Inductive definitions are natural to write and convenient to use because behind the scenes, theorem provers automatically generate proper definitions, which ensure that the introduction of the data type is a conservative extension [93]. Furthermore, numerous theorems are derived from the definitions available for the user and for the tactics [20, 93, 19].

As explained in Section 9.3.3, when proving completeness, all definitions and axioms of the mathematical structure have to be translated to the context of the corresponding model class, and have to be proved using the specification of the model class. The set of definitions and axioms should also include those that are implicitly generated for inductive structures by the theorem prover. However, the correctness of the implicit definitions and axioms follows from meta-theoretical results [93, 19] and are typically not derivable from the specification of the model class.

For instance, in Isabelle new functions are introduced and axiomatized for every inductive data type [19, 18]. These functions and axioms allow the automatic derivation of essential theorems, for instance, the induction principle. However, these implicit axioms are impossible to derive from the specification of model classes, since the functions that Isabelle implicitly introduces do not even exist in the model classes.

Furthermore, the most important implicitly derived theorems are not deducible from model-class specifications. Consider the induction principle for data type list, which states that for every property P the following holds:

$$(\, P\ \texttt{Nil}\ \wedge\ (\forall xs, x.\ P\ xs \Rightarrow P\ (\texttt{Cons}\ x\ xs))\,)\ \Leftrightarrow\ \forall xs.\ P\ xs$$

The induction principle cannot be expressed in JML^{--}, which is based on first-order logic and, thus, does not allow one to quantify over predicate P. Therefore, the principle cannot be directly derived from model-class specifications.

These problems indicate that the technique presented in Section 9.3 to show completeness is not sufficient to handle inductively defined types, because the proof would always fail for the implicitly generated definitions and axioms. In this chapter, we address this problem in two steps:

1. We capture the characteristic properties of inductive structures from which all implicitly generated definitions and axioms follow.
2. We pose proof obligations and constraints on specified mappings to ensure that the characteristic properties carry over to the corresponding model class.

The proof obligations are to be proven beside those prescribed by the technique presented in Chapter 9. In fact, these additional proof obligations can be seen as *implicit axiom obligations* that stem from the inductive nature of the target structure.

In the sequel, we focus on the handling of inductive data types. Inductive sets can be handled in a similar manner.

Model Class `JMLObjectSequence`. Throughout this chapter, we will demonstrate our handling of inductive data types by the mapping of model class `JMLObjectSequence` to Isabelle's inductive data type $(\alpha)list$, which is defined in the `HOL/List` theory. The mapping of the class and a number of methods is presented in Figure 12.1. Note that the parameterless constructor and method `insertFront` are specified to be **`constructing`**.

The class is specified by an equational theory and method specifications. Figure 12.1 presents an example equation and a method specification. Other specifications and the implementations of methods are omitted.

Proving the faithfulness of the specified mapping is to be performed the same way as described in Chapter 9. In the sequel, we only focus on the additional obligations that one has to prove for the model class due to the inductively defined target structure, $(\alpha)list$. The proofs of the additional obligations are available online [136].

Characteristic Properties

In the sequel, we only consider non-nested, non-mutually-recursive type definitions, which are sufficient to cover the mapping of many practically useful

```
mapped_to("Isabelle", "HOL/List", "α list");
immutable pure model class JMLObjectSequence {

    invariant (\forall JMLObjectSequence s2. s2 != null ==>
                (\forall Object e1. (\forall Object e2.
                  equational_theory(this, s2, e1, e2))));

    static pure boolean
      equational_theory(JMLObjectSequence s, JMLObjectSequence s2,
                        nullable Object e1, nullable Object e2)
        ensures \result <==> s.insertFront(e1).concat(s2).equals(
                               s.concat(s2).insertFront(e1));

    mapped_to("Isabelle", "Nil");
    constructing
    JMLObjectSequence();

    mapped_to("Isabelle", "Cons e Nil");
    JMLObjectSequence(nullable Object e);

    mapped_to("Isabelle", "nth this i");
    nullable Object itemAt(int i);

    mapped_to("Isabelle", "size this");
    int int_size();

    mapped_to("Isabelle", "elem mem this");
    boolean has(nullable Object elem);

    mapped_to("Isabelle", "this = obj");
    boolean equals(Object obj);

    mapped_to("Isabelle", "null this");
    boolean isEmpty()
      ensures \result == (int_size() == 0);

    mapped_to("Isabelle", "Cons item this");
    constructing
    JMLObjectSequence insertFront(nullable Object item);

    mapped_to("Isabelle", "this @ s2");
    JMLObjectSequence concat(JMLObjectSequence s2);

    // other specifications and methods omitted
}
```

Figure 12.1: Mapping and equational theory of `JMLObjectSequence`

model classes, in particular, they cover all model classes of the JML model library. An extension to nested, mutually-recursive definitions is possible by posing more complex proof obligations than the ones presented in this chapter.

The general form of a non-nested, non-mutually-recursive inductive data-type definition is [93]:

$$(\alpha_1, \ldots, \alpha_n)rty \ ::= \ \mathtt{C}_1 \ ty_1^1 \ \ldots \ ty_1^{k_1} \mid \ldots \mid \mathtt{C}_m \ ty_m^1 \ \ldots \ ty_m^{k_m} \qquad (12.1)$$

which defines type $(\alpha_1, \ldots, \alpha_n)rty$ with n type variables $\alpha_1, \ldots, \alpha_n$ where $n \geq 0$. The type has m constructors $\mathtt{C}_1$, ..., $\mathtt{C}_m$ where $m \geq 1$. Each constructor $\mathtt{C}_i$ takes k_i arguments where $k_i \geq 0$, and type expression ty_i^j for $1 \leq j \leq k_i$ either does not contain rty (i.e., is non-nested) or is equal to $(\alpha_1, \ldots, \alpha_n)rty$ (i.e., is recursive).

For example, the formal definition of type *list* is typically given as $(\alpha)list \ ::= \ \mathtt{Nil} \mid \mathtt{Cons} \ \alpha \ (\alpha)list$, where `Nil` takes no argument, the first argument of `Cons` is non-nested, and the second is recursive.

The data type defined in the form of (12.1) denotes the minimal set of all values that can be finitely generated using the constructors $\mathtt{C}_1$, ..., $\mathtt{C}_m$, where the constructors possess the so-called freeness properties: each constructor is injective, and two different constructors yield distinct values [93].

In the sequel, we show how the minimality of the denoted set and the freeness properties can be enforced on a model class.

Minimal Set of Finitely Constructable Values. Given an inductively defined data type S, we know that all values of the type are finitely constructable by the constructors of S. To make sure that this property also holds for model class M that is mapped to S, we require that there is a direct correspondence between the methods that are mentioned in the universe predicate of M and the constructors of S. More specifically, if method n is marked as **`constructing`** and the method takes one implicit and k explicit parameters, then we require that for all $t, x_1, \ldots, x_k$:
(1) $\Phi_M(t.n(x_1, \ldots, x_k))$ yields an application of $\mathtt{C}_i$ for some $1 \leq i \leq m$; and
(2) $\Phi_S(\Phi_M(t.n(x_1, \ldots, x_k))) = \widehat{n}(t, x_1, \ldots, x_k)$.

Note that the requirement ensures that the "reverse" translation of type constructors is standard.

Example 12.1. The above requirement holds for both operations of class `JMLObjectSequence` that are marked as **`constructing`**. For instance, for method `insertFront` we have:

$$\begin{aligned} \Phi_M(t.\mathtt{insertFront}(x)) &\equiv \mathtt{Cons} \ x \ t \\ \Phi_S(\Phi_M(t.\mathtt{insertFront}(x))) &\equiv \widehat{insertFront}(t, x) \end{aligned}$$

Note, however, that the requirement does not hold for the constructor that takes one argument, because the second condition does not hold:

$$\Phi_S(\Phi_M(\texttt{JMLObjectSequence}(x))) \equiv \widehat{insertFront}(\widehat{JMLObjectSequence}(), x)$$

Therefore, this (redundant) constructor may not be marked. As one would expect, the inductively defined universe predicate $U_{\widehat{M}}$ looks as follows:

$$\begin{aligned} U_{\widehat{M}}(x) \triangleq\ & \widehat{equals}(x, \widehat{JMLObjectSequence}()) \vee \\ & (\exists\, s, e.\ U_{\widehat{M}}(s) \wedge \widehat{equals}(x, \widehat{insertFront}(s, e))) \end{aligned} \tag{12.2}$$

□

Note that we do not require that every constructor of the data type is mapped to. That is, the data type might contain values that do not have counterparts in the model class—more precisely, in the set of values defined by the universe predicate. This apparent mismatch does not lead to unsoundness, because the properties that are implicitly derived from an inductive definition hold even if values constructable by some of the type constructors are never actually encountered.

Similarly, we do not require that all constructable model instances can be constructed by one of the methods that form the universe predicate. That is, the model class might contain instances that do not have counterparts in the data type. Again, this mismatch does not lead to unsoundness, because (1) the universe predicate determines the set of model instances over which the translation is defined, and (2) proof obligation (9.6) on page 147 ensures that model fields take values only from the defined universe of the corresponding model class.

No Loops Requirement. Despite our attempt to make a close relationship between the universe predicate and the type constructors, we cannot yet guarantee that the construction of every model instance denoted by the universe predicate is finite. In order to ensure that, we need to prove a property for the model class that corresponds to the so-called *no loops* property of inductive types. Namely, for every non-empty sequence of nested type-constructor applications and value s of the data type, the following holds:

$$s \neq \mathtt{C}_i(\ldots, \mathtt{C}_j(\ldots, s, \ldots), \ldots) \tag{12.3}$$

Note that the term on the right-hand side of the inequality is an arbitrary sequence of constructor applications. Thus, proving the property directly for model classes would lead to an infinite set of proof obligations. Therefore, we identify some measure μ in the data type for which the following three properties hold.

First, any two values of the type with different measures are distinct:

$$\forall\, s_1,\, s_2.\ \mu(s_1) \neq \mu(s_2) \ \Rightarrow\ s_1 \neq s_2 \tag{12.4}$$

Second, for every constructor $\mathtt{C}_i$ that has at least one recursive argument, an application of the constructor increases the measure. For simplicity, the property is formally given for a constructor with one recursive and one non-recursive argument:

$$\forall\, s,\, e.\ \mu(s) < \mu(\mathtt{C}_i(s, e)) \tag{12.5}$$

By a simple inductive argument on the number of constructor applications on the right-hand side of (12.3), one can trivially show that these two properties together imply the absence of loops.

Third, the measure must be such that it can be unconditionally translated to the context of the model class. That is, translation function Φ_S must be applicable on the measure for all arguments it may take. This requirement is necessary to make sure that the translation of properties (12.4) and (12.5) is standard.

Given a measure that fulfills these three properties, the no loops property has to be proven by stating proof obligations in the model theory that correspond to the Φ_S-translation of properties (12.4) and (12.5).

Example 12.2. In data type $(\alpha)list$ (and in every other inductively defined data type in Isabelle [107]), a proper measure is function *size*, which yields the size of a given list. It can be trivially proven that properties (12.4) and (12.5) hold for the function, and, according to the specified mappings in Figure 12.1, the function directly maps back to method `int_size` of class `JMLObjectSequence`.

The proof obligations stemming from the instantiation and translation of properties (12.4) and (12.5) are the following, respectively:

$$\forall\, s_1,\, s_2.\ U_{\widehat{M}}(s_1) \Rightarrow U_{\widehat{M}}(s_2) \Rightarrow \widehat{int\_size}(s_1) \neq \widehat{int\_size}(s_2) \ \Rightarrow\ \neg\widehat{equals}(s_1, s_2)$$

$$\forall\, s,\, e.\ U_{\widehat{M}}(s) \Rightarrow \widehat{int\_size}(s) < \widehat{int\_size}(\widehat{insertFront}(s, e))$$

Note that there is only one proof obligation that corresponds to property (12.5), because `Nil` and the corresponding parameterless model constructor do not take recursive arguments.

In our case study, both proof obligations were proven by *auto* that we provided with hints on which axioms of the model theory to use. □

Given the no loops requirement and the requirements that ensure the close correspondence of universe predicates and type constructors, we have achieved our first main goal: the translation between the model class and the data type is defined over model instances that have a one-to-one correspondence to values that can be finitely constructed by the type constructors. The finiteness of the construction and the absence of values not constructable by type constructors (so-called *junk* elements) is guaranteed for the model class by the inductive nature of the universe predicate and the no loops requirement.

It remains to show that the freeness properties of type constructors carry over to the model class.

Injectivity of Constructors. For a type defined in the form of (12.1), the property expresses that for every constructor $\mathtt{C}_i$ and their parameters, the following holds:

$$\mathtt{C}_i\ x_i^1\ \ldots\ x_i^{k_i}\ =\ \mathtt{C}_i\ y_i^1\ \ldots\ y_i^{k_i}\ \Leftrightarrow\ x_i^1 = y_i^1\ \wedge\ \ldots\ \wedge\ x_i^{k_i} = y_i^{k_i}$$

The property can be enforced on model classes by instantiating the formula for each type constructor that participates in the translation, and by translating the resulting formula instances by function Φ_S. Finally, the translated formulas have to be stated as lemmas in and proven by the model theory.

Example 12.3. When proving the property for class `JMLObjectSequence`, we need to consider the injectivity of the two constructors of type $(\alpha)list$. Proving the injectivity property associated with `Nil` is trivial since the type constructor does not take any parameter. Since the corresponding constructor `JMLObjectSequence()` does not take parameters either and is assumed to be deterministic, the property also trivially holds for the constructor.

The property for `Cons` is the following:

$$\forall\, s_1, s_2, e_1, e_2.\ \mathtt{Cons}\ e_1\ s_1 = \mathtt{Cons}\ e_2\ s_2\ \Leftrightarrow\ e_1 = e_2\ \wedge\ s_1 = s_2$$

The translation of the formula yields the proof obligation:

$$\begin{aligned}\forall\, s_1,\ s_2,\ e_1,\ e_2.\ & U_{\widehat{M}}(s_1) \Rightarrow U_{\widehat{M}}(s_2) \Rightarrow \\ & (\,\widehat{equals}(\widehat{insertFront}(s_1, e_1),\ \widehat{insertFront}(s_2, e_2))\ \Leftrightarrow \\ & \quad e_1 = e_2\ \wedge\ \widehat{equals}(s_1, s_2)\,)\end{aligned}$$

The proof obligation was discharged by giving hints to the *auto* tactic on which axioms of the model theory to use, by a quantifier instantiation, and by a case split. □

Distinctness of Constructors. For a type defined in the form of (12.1), the property expresses that for every pair of constructors $\mathtt{C}_i$ and $\mathtt{C}_j$, where $i \neq j$, and for all parameters of $\mathtt{C}_i$ and $\mathtt{C}_j$, the following holds:

$$\mathtt{C}_i\ x_i^1\ \ldots\ x_i^{k_i}\ \neq\ \mathtt{C}_j\ y_j^1\ \ldots\ y_j^{k_j}$$

The property can be enforced on model classes by instantiating the formula for each pair of constructors that participate in the mapping. The translation and proof method of the formula instances is the same as for injectivity.

Example 12.4. Type $(\alpha)list$ has two constructors, thus it has only one distinctness property: for every list s and element e, $\mathtt{Nil} \neq \mathtt{Cons}\ e\ s$ holds. The Φ_S-translation of the formula yields the proof obligation:

$$\forall\, s,\, e.\ U_{\widehat{M}}(s) \Rightarrow \neg\ \widehat{equals}(JMLObjectSequence(),\ \widehat{insertFront}(s, e))$$

The proof obligation was discharged by giving hints to the *auto* tactic on which axioms of the model theory to use. □

Guideline. As mentioned earlier, if the source and the target universes do not match and the universe predicate is defined inductively, then faithfulness proofs become rather difficult.

The situation is even worse when the target structure is inductively defined: according to the requirement on the relationship between the universe predicate and the type constructors (see page 168), the universe predicate cannot be defined to be the trivial one, even if the source and the target universes match. This leads to superfluous proof efforts.

Therefore, we define conditions, which guarantee that the universe of the model class and the related structure match. If the conditions hold, then the universe predicate can be defined to be *true*, otherwise it has to be defined as before.

The conditions are the following: (1) *every* type constructor $\mathtt{C}_i$ is related to a *constructor* of the model class such that the requirement presented on page 168 holds; and (2) every other constructor of the model class, if any, is redundant: it yields values that can be constructed by the constructors that are related to type constructors. The latter condition is to be proven by proof obligations of the form of formula (9.4) presented on page 142.

Example 12.5. For model class `JMLObjectSequence`, the universe predicate can be considered to be *true* if (1) its interface is modified such that method `insertFront` is turned into a constructor, and (2) one can prove the following formula for the "redundant" constructor `JMLObjectSequence(e)`:

$$\forall\, x_1, y_1, y_2.\ x_1 = y_2 \ \wedge\ \widehat{equals}(y_1, \widehat{JMLObjectSequence}()) \ \Rightarrow \\ \widehat{equals}(\widehat{JMLObjectSequence}(x_1), \widehat{JMLObjectSequence}(y_1, y_2))$$

where term $\widehat{JMLObjectSequence}(y_1, y_2)$ corresponds to a call to the constructor that replaces method `insertFront`. □

Note that proving the above formula is significantly simpler than, for instance, proving the function symbol obligation with universe predicate (12.2) given on page 169.

The Fruit of Our Labor. The additional requirements and proof obligations presented in this chapter allow us to map model classes to inductively defined types in a sound way, that is, without the danger of endowing the class with properties that it does not actually possess.

In fact, properties of inductively defined types (in particular, the induction principle) may be added to the model theory *after* the additional requirements and proof obligations have been shown for the model class. These properties may assist in accomplishing the rest of the completeness proof.

Note that adding, for instance, the induction principle to the model theory makes the theory stronger than the specification of the model class. This might seem undesired, however, the model theory is only used for the completeness proof, and the theory that is used for the consistency proof and eventually for the verification of client code is the target theory S, which does contain the induction principle.

Chapter 13

Related Work

Theory Interpretation. The interpretation of first-order theories is a well-established technique and has been described in several textbooks on mathematical logic. The technique has already been used for formal program development. For instance, Levy applied theory interpretation to formally show the correctness of compiler implementations [88]. Ergo, the theorem prover of the Cogito system [104] applies theory interpretation to maximize theory reuse [57]. A concrete instance of the interpretation in Ergo is the development of real numbers by Shield *et al.* [124].

We are not aware of applications of the theory in the realm of one- and two-tiered specification languages.

Theorem-Prover Specific Symbols. The idea of using function symbols that are understood by the back-end theorem prover directly on the specification level was already present in ESC/Java [47]. The special specification construct `\dttfsa` (*Damn The Torpedos, Full Speed Ahead!*) allows users to apply function symbols that are defined directly in Simplify, the theorem prover of ESC/Java. While the construct is a powerful means for specification, one has to be careful with its usage since the meaning of the symbols are hidden on the specification level. In particular, the verification system does not give support for showing that the definitions are consistent.

Similarly, for specification and verification purposes, the Caduceus tool [44] allows one to declare types and predicates as well as to define or axiomatize these predicates on the source level. One can also define "hybrid" predicates, predicates that refer both to elements of the program and elements of these specification-only types and predicates. Definitions of predicates can also be postponed on the source level and given directly in Coq, the back-end prover of the tool. This concept eases the task of specifying and verifying programs since, for instance, it prevents the use of method calls in specifications and leads to definitions that are more suitable for provers than JML specifications. Case studies demonstrate the power of this approach

[45, 64]. The drawback of the approach is the absence of a consistency proof for definitions and axioms given on the source or prover level [90]. This might lead to soundness issues.

Model Classes. Schoeller introduces model classes to Eiffel as a solution for writing more complete and abstract contracts [121]. He roughly sketches the idea of the faithful mapping of model classes to mathematical structures, however, no formal or semi-formal details are given on how one would prove faithfulness.

Schoeller *et al.* developed a model library for Eiffel [123, 122]. They address the faithfulness issue by equipping methods of model classes with specifications that directly correspond to axioms and theorems taken from mathematical textbooks. A shortcoming of this approach is that the resulting model library has to follow exactly the structure of the mimicked theory. This limits the design decisions one can make when composing the model library and it is unclear how one can support multiple theorem provers. Our approach allows more flexibility in the construction of model classes and libraries by using `mapped_to` clauses that can go beyond direct mappings since arbitrary terms of the target context can be specified. In turn, our approach requires one to prove faithfulness of the mapping.

Charles [29] proposes the introduction of the `native` keyword to JML in the context of work on the program verifier Jack [26]. The keyword can be attached to methods with a similar meaning to ESC/Java's `\dttfsa` construct: methods marked as `native` introduce uninterpreted function symbols, and their definitions can be directly given on the level of Coq, the back-end prover of Jack. Charles carries the idea over to model classes: the `native` keyword may also be attached to types with the meaning that such types get mapped to corresponding Coq data types.

This approach differs from ours in two ways. First, our approach ensures faithfulness of the mapping. There is no attempt to do so in the work of Charles. Second, the `mapped_to` clause we propose in this thesis allows one to specify the mapping on the specification language level. Furthermore, properties of model classes are specified in JML, which typically provides easier understanding (for programmers) of the semantics than definitions given directly on the level of a theorem prover.

Leavens *et al.* [76] identify the problem of specifying model classes as a research challenge. They propose four possible solution approaches and summarize the open problems for each of them. The first and second solution approaches advocate the use of equational theories and method specifications, respectively. The issues regarding the verification of the implementations of model-class methods are highlighted for both approaches. These issues are not considered in this thesis since our approach focuses on the consistency and completeness of the *specification* of model classes.

The third solution of Leavens *et al.* considers automatic translations be-

tween model classes and mathematical structures, and the authors argue why such translations are difficult. We deal with these problems by specifying the mapping manually and proving faithfulness of the mapping. One of the problems mentioned by Leavens *et al.* is how to deal with model class specifications that invoke methods of the program, in particular, `Object.equals`. As discussed in Section 10.4, handling equality of elements can be solved by the duplication of target theories. The fourth solution approach of Leavens *et al.* is similar to the work by Schoeller and Charles, discussed above.

Runtime Approaches. There is a large body of work on the runtime discovery and checking of algebraic specifications. We give an incomplete list of references and refer the reader to [60] for a more detailed account on previous work. Sankar [119], Antoy and Hamlet [5], and Nunes *et al.* [108] developed systems that allow one to test the implementation of algebraic data types against their implementations. Henkel and Diwan [60, 61] developed an approach for the discovery of algebraic specifications as well as for their debugging. These approaches are orthogonal to ours. First, they inspect the relation between the specification and implementation of data types. Our approach inspects the relation between the specifications of two mathematical structures. Second, the listed approaches are dynamic, that is, based on executions of the inspected data types. Thus, these approaches cannot guarantee that the result of the specification discovery and checking is sound. Our approach is based on static verification techniques.

Chapter 14

Conclusion

We conclude the thesis with a brief summary of the main contributions and a brief discussion of areas where the presented techniques could be further improved.

14.1 Contributions

Abstraction techniques are not only indispensable in state-of-the-art programming languages, but also in state-of-the-art specification languages. Abstraction techniques used in specification languages facilitate information hiding as well as the readability and maintainability of specifications.

This thesis focused on two commonly used means of abstraction: pure methods and model classes. Although the use of these means were well-understood for the purpose of specification, there was little support for their sound translation to theorem provers, which is crucial for the verification of programs.

Pure Methods. The contribution of our work on the encoding and axiomatization of pure methods is two-fold:

First, we demonstrated that the state changes pure methods potentially make are observable in specification expressions. Therefore, encodings of pure methods either have to model such state changes explicitly or have to find ways to make them non-observable.

We sketched the formal details of an encoding with explicit modeling of state changes, and argued that such an encoding is impractical for program verification. Therefore, we proposed a more practical encoding that handles weakly-pure methods as if they were strongly pure. We argued for the correctness of the encoding by performing a rigorous analysis of the consequences of omitting heap changes for the small, but expressive specification language JML^{--}.

Second, we proposed a technique for the axiomatization of pure-method specifications. The technique applies existing approaches for checking satisfiability, well-definedness, and well-foundedness of specifications, as well as for handling dependencies between specification elements. However, to the best of our knowledge, our approach is the first one that puts all these approaches together to yield a satisfying solution for a one-tiered object-oriented specification language.

The feasibility of the technique was demonstrated by an implementation in the Spec# verification system. The implementation required practical considerations in order to work with a fully automated back-end theorem prover.

Model Classes. Our contribution is a technique that allows one to formally prove that the specified mapping of a model class to a mathematical structure is semantically faithful. The technique is based on the theory of theory interpretation and takes the specifications of model classes into account.

The mapping of model classes to existing theories of theorem provers is not a novel idea of ours. However, previous approaches did not take specifications into account, thereby, did not ensure that there was a semantic relationship between the mapped entities. Therefore, such approaches could not guarantee that the specified and the actually verified properties indeed corresponded to each other.

Our approach leads to better specifications for model classes by ensuring their (relative) consistency and completeness. The identification and checking of redundant specifications further improves the quality of the specifications.

The usefulness of the approach was demonstrated by a case study on two of the most basic model classes of the JML library. In one of the two model classes, the case study revealed an incorrect specification and identified a precise relation between the equational theory and the method specifications of the model class.

The techniques developed both for the handling of pure methods and for the mapping of model classes are independent of the specific specification language and theorem prover to which they are applied. Therefore, the techniques can be readily adapted by program provers that build upon state-of-the-art specification languages and theorem provers, such as Eiffel, JML, or Spec#, and Isabelle, PVS, or Coq, respectively.

14.2 Future Work

Theory. The correctness of the simplified encoding function has not been formally proven in Chapter 4. Such a proof is far from trivial. It requires

the formalization of two evaluation functions over JML^{--} expressions: one that takes heap changes into account and one that does not. Correctness of the simplified encoding would be justified if the equivalence of the two evaluation functions could be proven under the assumption that the prescribed restrictions on JML^{--} specifications are not violated.

A serious limitation posed by our axiomatization technique is that invariants may not contain pure-method calls. A solution to this problem is highly desirable since the use of method calls in invariants is just as common and natural as in method specifications.

Further research is needed to see how recursion measures for pure methods that traverse object structures can be conveniently specified, and how the well-foundedness of such measures can be proven. Recent research results on the verification of common data structures may provide hints or answers.

Our technique for the faithful mapping of model classes could be extended in different ways. For instance, as mentioned in Chapter 10, the handling of ghost fields is not worked out in this thesis. Furthermore, our technique is based on standard translations, which do not allow a method to be mapped in different ways under different conditions. Such conditional mappings may be useful in practice, for instance, in Example 8.2 on page 117, where depending on whether the value of parameter `e` is **`null`** or not, constructor `JMLObjectSet(e)` should be mapped either to the empty set or to the singleton set, respectively. Such extensions would make our approach more flexible, thereby capable of covering specification elements that refer to ghost fields, and to cover model classes with more complex behavior.

Another issue for which this thesis did not give a particularly satisfying solution is the handling of equality of elements of model classes. A solution that did not require the duplication of theories would be much more favorable.

Tools. The implementation of our axiomatization technique in the Spec# system is rather prototypical. More development efforts should be spent on it to make it ready for public use.

Future work remains to provide tool support for the technique proposed for the faithful mapping of model classes. Tools could support the type-checking of `mapped_to` clauses, the (partial) generation of proof scripts for the faithfulness proofs, the checking and identification of redundant specifications, and the actual translation of model-method calls during the static verification of programs.

Experience. In order to give a fair judgement on the usability and practicality of the presented techniques, more examples and case studies should be developed. Our implementation of the pure-method axiomatization technique has not yet been put to the test by larger examples. Although it works

well for smaller programs, an extensive case study could reveal the rough edges of our technique and implementation.

Although we have done case studies on our approach to the faithful mapping of model classes, further case studies would be needed (1) to provide a small but powerful library with model classes whose mapping to a number of commonly used theorem provers has been proven faithful, and (2) to see if the approach is necessary to extend, for instance, with conditional mappings.

The field of specification and verification of object-oriented programs has seen an immense progress in the last two decades. Still, there remains much research and engineering to be done in the field to provide programmers with methodologies and tools that adequately support them in developing correct programs. Our hope is that this thesis brings the field one step closer to that level.

Appendix A

Interpretation and Well-definedness of FOL

The three-valued interpretation of first-order terms and formulas is defined in Figure A.2. The well-definedness conditions that $\mathcal{L}$ yields corresponds to this interpretation.

The interpretation of functions and predicates is *strict*: an undefined argument makes their interpretation to be undefined. **dom** yields the domain of the interpretation of functions. Predicates are considered to be total on well-defined arguments. The interpretation of conjunction and disjunction is according to McCarthy's semantics (see Section 5.1.2). The interpretation of quantifiers is *strict*: they are defined if and only if quantified expressions are defined for all instantiations.

The definition of operator $\mathcal{L}$ over terms and formulas is the following, where Var is a variable, f is a function symbol, P is a predicate, e_i are terms, and ϕ, ϕ_1, and ϕ_2 are formulas.

$$
\begin{array}{lcl}
\mathcal{L}(Var) & \triangleq & \mathit{true} \\
\mathcal{L}(f(e_1, \ldots, e_n)) & \triangleq & \bigwedge_{i=1}^{n} \mathcal{L}(e_i) \ \wedge \ \delta(f)(e_1, \ldots, e_n) \\
\mathcal{L}(P(e_1, \ldots, e_n)) & \triangleq & \bigwedge_{i=1}^{n} \mathcal{L}(e_i) \\
\mathcal{L}(\mathit{true}) & \triangleq & \mathit{true} \\
\mathcal{L}(\mathit{false}) & \triangleq & \mathit{true} \\
\mathcal{L}(\neg\phi) & \triangleq & \mathcal{L}(\phi) \\
\mathcal{L}(\phi_1 \Rightarrow \phi_2) & \triangleq & \mathcal{L}(\phi_1) \wedge (\phi_1 \Rightarrow \mathcal{L}(\phi_2)) \\
\mathcal{L}(\phi_1 \wedge \phi_2) & \triangleq & \mathcal{L}(\phi_1) \wedge (\phi_1 \Rightarrow \mathcal{L}(\phi_2)) \\
\mathcal{L}(\phi_1 \vee \phi_2) & \triangleq & \mathcal{L}(\phi_1) \wedge (\neg\phi_1 \Rightarrow \mathcal{L}(\phi_2)) \\
\mathcal{L}(\forall\, x.\ \phi) & \triangleq & \forall\, x.\, \mathcal{L}(\phi) \\
\mathcal{L}(\exists\, x.\ \phi) & \triangleq & \forall\, x.\, \mathcal{L}(\phi)
\end{array}
$$

Figure A.1: Definition of the $\mathcal{L}$ operator over terms and formulas.

$$
\begin{array}{lcl}
[\,\mathbf{v}\,]^3_{\mathbf{M}}\theta & \triangleq & \theta(\mathbf{v}) \text{ where } \mathbf{v} \text{ is a variable} \\
[\,f(\mathbf{t}_1,\ldots,\mathbf{t}_n)\,]^3_{\mathbf{M}}\theta & \triangleq & \begin{cases} \mathbf{I}(f)([\mathbf{t}_1]^3_{\mathbf{M}}\theta,\ldots,[\mathbf{t}_n]^3_{\mathbf{M}}\theta), \\ \qquad\qquad \text{if } [\mathbf{t}_1]^3_{\mathbf{M}}\theta \neq \bot,\ldots,\ [\mathbf{t}_n]^3_{\mathbf{M}}\theta \neq \bot \text{ and} \\ \qquad\qquad \langle [\mathbf{t}_1]^3_{\mathbf{M}}\theta,\ldots,[\mathbf{t}_n]^3_{\mathbf{M}}\theta\rangle \in \mathbf{dom}(\mathbf{I}(f)) \\ \bot, \qquad\qquad \text{otherwise} \end{cases} \\
[\,P(\mathbf{t}_1,\ldots,\mathbf{t}_n)\,]^3_{\mathbf{M}}\theta & \triangleq & \begin{cases} \mathbf{true}, & \text{if } [\mathbf{t}_1]^3_{\mathbf{M}}\theta \neq \bot,\ldots,[\mathbf{t}_n]^3_{\mathbf{M}}\theta \neq \bot \text{ and} \\ & \quad \mathbf{I}(P)([\mathbf{t}_1]^3_{\mathbf{M}}\theta,\ldots,[\mathbf{t}_n]^3_{\mathbf{M}}\theta) = \mathbf{true} \\ \mathbf{false}, & \text{if } [\mathbf{t}_1]^3_{\mathbf{M}}\theta \neq \bot,\ldots,[\mathbf{t}_n]^3_{\mathbf{M}}\theta \neq \bot \text{ and} \\ & \quad \mathbf{I}(P)([\mathbf{t}_1]^3_{\mathbf{M}}\theta,\ldots,[\mathbf{t}_n]^3_{\mathbf{M}}\theta) = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\mathit{true}\,]^3_{\mathbf{M}}\theta & \triangleq & \mathbf{true} \\
[\,\mathit{false}\,]^3_{\mathbf{M}}\theta & \triangleq & \mathbf{false} \\
[\,\neg\varphi\,]^3_{\mathbf{M}}\theta & \triangleq & \begin{cases} \mathbf{true}, & \text{if } [\varphi]^3_{\mathbf{M}}\theta = \mathbf{false} \\ \mathbf{false}, & \text{if } [\varphi]^3_{\mathbf{M}}\theta = \mathbf{true} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\varphi \Rightarrow \phi\,]^3_{\mathbf{M}}\theta & \triangleq & \begin{cases} \mathbf{true}, & \text{if } [\varphi]^3_{\mathbf{M}}\theta = [\phi]^3_{\mathbf{M}}\theta = \mathbf{true}, \text{ or } [\varphi]^3_{\mathbf{M}}\theta = \mathbf{false} \\ \mathbf{false}, & \text{if } [\varphi]^3_{\mathbf{M}}\theta = \mathbf{true} \text{ and } [\phi]^3_{\mathbf{M}}\theta = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\varphi \wedge \phi\,]^3_{\mathbf{M}}\theta & \triangleq & \begin{cases} \mathbf{true}, & \text{if } [\varphi]^3_{\mathbf{M}}\theta = [\phi]^3_{\mathbf{M}}\theta = \mathbf{true} \\ \mathbf{false}, & \text{if } [\varphi]^3_{\mathbf{M}}\theta = \mathbf{false} \text{ or} \\ & \quad [\varphi]^3_{\mathbf{M}}\theta \neq \bot \text{ and } [\phi]^3_{\mathbf{M}}\theta = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\varphi \vee \phi\,]^3_{\mathbf{M}}\theta & \triangleq & \begin{cases} \mathbf{true}, & \text{if } [\varphi]^3_{\mathbf{M}}\theta = \mathbf{true} \text{ or} \\ & \quad [\varphi]^3_{\mathbf{M}}\theta \neq \bot \text{ and } [\phi]^3_{\mathbf{M}}\theta = \mathbf{true} \\ \mathbf{false}, & \text{if } [\varphi]^3_{\mathbf{M}}\theta = [\phi]^3_{\mathbf{M}}\theta = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\forall x.\ \varphi\,]^3_{\mathbf{M}}\theta & \triangleq & \begin{cases} \mathbf{true}, & \text{if for all } \mathbf{l} \in \mathbf{A},\ [\varphi]^3_{\mathbf{M}}\theta[x \leftarrow \mathbf{l}] = \mathbf{true} \\ \mathbf{false}, & \text{if for all } \mathbf{l} \in \mathbf{A},\ [\varphi]^3_{\mathbf{M}}\theta[x \leftarrow \mathbf{l}] \neq \bot \text{ and there} \\ & \quad \text{exists } \mathbf{l} \in \mathbf{A} \text{ such that} [\varphi]^3_{\mathbf{M}}\theta[x \leftarrow \mathbf{l}] = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases} \\
[\,\exists x.\ \varphi\,]^3_{\mathbf{M}}\theta & \triangleq & \begin{cases} \mathbf{true}, & \text{if for all } \mathbf{l} \in \mathbf{A},\ [\varphi]^3_{\mathbf{M}}\theta[x \leftarrow \mathbf{l}] \neq \bot \text{ and there} \\ & \quad \text{exists } \mathbf{l} \in \mathbf{A} \text{ such that} [\varphi]^3_{\mathbf{M}}\theta[x \leftarrow \mathbf{l}] = \mathbf{true} \\ \mathbf{false}, & \text{if for all } \mathbf{l} \in \mathbf{A},\ [\varphi]^3_{\mathbf{M}}\theta[x \leftarrow \mathbf{l}] = \mathbf{false} \\ \bot, & \text{otherwise} \end{cases}
\end{array}
$$

Figure A.2: McCarthy's interpretation of FOL terms and formulas.

Appendix B

Proof of Theorem 5.2

Theorem 5.2. *For any* JML^{--} *expression* E*, and stores* OS' *and* OS*, the following equation holds:*

$$Df(E, OS', OS) \;\equiv\; \mathcal{L}(\gamma(E, OS', OS))$$

Proof. The proof runs by induction on the structure of JML^{--} expressions. For the proof, we may assume the induction hypothesis: for every sub-expression F of E and stores OS' and OS, the following holds:

$$Df(F, OS', OS) \;\equiv\; \mathcal{L}(\gamma(F, OS', OS))$$

Base case. In the base case, E is some parameter name or literal.
From the definition of Df in Figure 5.2 on page 74, we know that:
$Df(E, OS', OS) \equiv true$
By the definition of γ in Figure 4.3 on page 50, we get that:
$\mathcal{L}(\gamma(E, OS', OS)) \equiv \mathcal{L}(FOL(E))$
From the definition of function FOL in Figure 4.4 on page 51, we can deduce that parameter names and literals are encoded by variables (*e.g.*, *param*), total (constant) function symbols (*e.g.*, *null*), and symbols *true* and *false*. For all these cases, the definition of $\mathcal{L}$ yields *true* (see Figure A.1). Hence, $\mathcal{L}(FOL(E)) \equiv true$.

Induction step. We consider the compound expressions of JML^{--}.

• E is expression F `&&` G.
By definition,
$Df(F \texttt{ \&\& } G, OS', OS) \equiv$
$Df(F, OS', OS) \wedge (\gamma(F, OS', OS) \Rightarrow Df(G, OS', OS))$.
By the definition of γ and $\mathcal{L}$, we can deduce that
$\mathcal{L}(\gamma(F \texttt{ \&\& } G, OS', OS)) \;\equiv\; \mathcal{L}(\gamma(F, OS', OS) \wedge \gamma(G, OS', OS)) \;\equiv$
$\mathcal{L}(\gamma(F, OS', OS)) \wedge (\gamma(F, OS', OS) \Rightarrow \mathcal{L}(\gamma(G, OS', OS)))$

By the induction hypothesis on F and G, we get the desired property.

• The cases of logical operators `||`, `==>`, and `!`, as well as the cases of total operators `==`, `!=`, `<`, `<=`, `+`, `-`, and `*` are analogous to the previous case.

• E is expression $F \otimes G$, where $\otimes$ is one of the partial operators `/` or `%`.
By definition,

$Df(F \otimes G, OS', OS) \equiv$
$Df(F, OS', OS) \wedge Df(G, OS', OS) \wedge \gamma(G, OS', OS) \neq 0.$

By the definition of γ, we can derive that

$\mathcal{L}(\gamma(F \otimes G, OS', OS)) \equiv \mathcal{L}(\gamma(F, OS', OS)\ \ FOL(\otimes)\ \ \gamma(G, OS', OS))$

Operator $FOL(\otimes)$ corresponds to the partial function symbol *div* or *mod*. Applying the definition of $\mathcal{L}$ with the defined domain restriction δ_γ in Figure 5.3 on page 76, we can further deduce that

$\mathcal{L}(\gamma(F, OS', OS)\ \ FOL(\otimes)\ \ \gamma(G, OS', OS)) \equiv$
$\mathcal{L}(\gamma(F, OS', OS))\ \wedge\ \mathcal{L}(\gamma(G, OS', OS))\ \wedge\ \gamma(G, OS', OS) \neq 0$

By the induction hypothesis on F and G, we get the desired property.

• E is expression F`.g`.
By definition, $Df(F\texttt{.g}, OS', OS) \equiv Df(F, OS', OS) \wedge \gamma(F, OS', OS) \neq null$.
By the definition of γ we get that

$\mathcal{L}(\gamma(F\texttt{.g}, OS', OS)) \equiv \mathcal{L}(\ OS'(loc(\gamma(F, OS', OS), g))\ )$

Applying $\mathcal{L}$ on the total lookup function and on the partial *loc* function with its defined domain restriction in δ_γ, we get that

$\mathcal{L}(\ OS'(loc(\gamma(F, OS', OS), g))\ )\ \equiv\ \mathcal{L}(\ loc(\gamma(F, OS', OS), g)\ ) \equiv$
$\mathcal{L}(\gamma(F, OS', OS))\ \wedge\ \mathcal{L}(g)\ \wedge\ \gamma(F, OS', OS) \neq null\ \equiv$
$\mathcal{L}(\gamma(F, OS', OS))\ \wedge\ \gamma(F, OS', OS) \neq null$

By the induction hypothesis on F, we get the desired property .

• E is expression F`.m(`G`)`. For simplicity `m` has one formal parameter `param`.
By definition,

$Df(F\texttt{.m(}G\texttt{)}, OS', OS)\ \equiv$
$\quad Df(F, OS', OS)\ \wedge\ Df(G, OS', OS)\ \wedge\ \gamma(F, OS', OS) \neq null\ \wedge$
$\quad Df(\mathrm{pre}^{\mathtt{T}}_{\mathtt{m}}[F/\mathbf{this}, G/\mathtt{param}], OS', OS)\ \wedge$
$\quad \gamma(\mathrm{pre}^{\mathtt{T}}_{\mathtt{m}}[F/\mathbf{this}, G/\mathtt{param}], OS', \_)$

By the definition of γ on method calls and $\mathcal{L}$ with domain restriction δ_γ on function applications, we get that

$\mathcal{L}(\gamma(F\texttt{.m(}G\texttt{)}, OS', OS))\ \equiv$
$\mathcal{L}(\widehat{m}(\gamma(F, OS', OS), \gamma(G, OS', OS), OS'))\ \equiv$
$\mathcal{L}(\gamma(F, OS', OS))\ \wedge\ \mathcal{L}(\gamma(G, OS', OS))\ \wedge\ \mathcal{L}(OS')\ \wedge$
$\gamma(F, OS', OS) \neq null\ \wedge\ Df(\mathrm{pre}^{\mathtt{T}}_{\mathtt{m}}[F/\mathbf{this}, G/\mathtt{param}], OS', OS)\ \wedge$
$\gamma(\mathrm{pre}^{\mathtt{T}}_{\mathtt{m}}[F/\mathbf{this}, G/\mathtt{param}], OS', \_) \equiv$
$\mathcal{L}(\gamma(F, OS', OS))\ \wedge\ \mathcal{L}(\gamma(G, OS', OS))\ \wedge\ \gamma(F, OS', OS) \neq null\ \wedge$

$Df(\mathrm{pre}^{\mathtt{T}}_{\mathtt{m}}[F/\mathtt{this}, G/\mathtt{param}], OS', OS) \ \wedge \ \gamma(\mathrm{pre}^{\mathtt{T}}_{\mathtt{m}}[F/\mathtt{this}, G/\mathtt{param}], OS', \_)$

By the induction hypothesis on F and G, we get the desired property.

• The case of constructor call **new** `C(`F`)` is analogous to the previous case.

• E is expression `\old(`F`)`.
By definition, $Df(\backslash\mathtt{old}(F), OS', OS) \equiv Df(F, OS, OS)$ and
$\mathcal{L}(\gamma(\backslash\mathtt{old}(F), OS', OS)) \equiv \mathcal{L}(\gamma(F, OS, OS))$
By the induction hypothesis on F with $OS = OS'$, we get the property.

• E is expression `\fresh(`F`)`.
By definition, $Df(\backslash\mathtt{fresh}(F), OS', OS) \equiv Df(F, OS', OS)$.
By the definition of γ, we get that
$\mathcal{L}(\gamma(\backslash\mathtt{fresh}(F), OS', OS)) \ \equiv$
$\mathcal{L}(\ \neg alive(\gamma(F, OS', OS), OS) \ \wedge \ \gamma(F, OS', OS) \neq null\ )$
Applying the definition of $\mathcal{L}$ on the conjunct, we get the following after trivial simplifications:
$\mathcal{L}(\gamma(F, OS', OS)) \wedge (\neg alive(\gamma(F, OS', OS), OS') \ \Rightarrow \ \mathcal{L}(\gamma(F, OS', OS)))$
which is equivalent to $\mathcal{L}(\gamma(F, OS', OS))$ as for any proposition A and B equivalence $A \wedge (B \Rightarrow A) \equiv A$ holds.
Thus, by the induction hypothesis on F, we get the desired property.

• E is expression `(\forall` $T\ x.\ F$`)`.
The proof is trivial for quantification over primitive types. We show the proof for the more complicated case when T is of reference type.
By definition,
$Df(\mathtt{(\backslash forall}\ T\ x.\ F\mathtt{)}, OS', OS) \equiv$
$\forall\ x.\ allocT(x, OS', T) \ \Rightarrow \ Df(F, OS', OS)$
By the definition of γ and $\mathcal{L}$, we can deduce that
$\mathcal{L}(\gamma(\mathtt{(\backslash forall}\ T\ x.\ F\mathtt{)}, OS', OS)) \ \equiv$
$\mathcal{L}(\ \forall\ x.\ allocT(x, OS', T) \ \Rightarrow \ \gamma(F, OS', OS)\ ) \ \equiv$
$\forall\ x.\ \mathcal{L}(\ allocT(x, OS', T) \ \Rightarrow \ \gamma(F, OS', OS)\ ) \ \equiv$
$\forall\ x.\ \mathcal{L}(allocT(x, OS', T)) \wedge (\ allocT(x, OS', T) \Rightarrow \mathcal{L}(\gamma(F, OS', OS))\ ) \ \equiv$
$\forall\ x.\ allocT(x, OS', T) \ \Rightarrow \ \mathcal{L}(\gamma(F, OS', OS))$
By the induction hypothesis on F, we get the desired property.

• E is expression `(\exists` $T\ x.\ F$`)`.
The proof is analogous to the previous case. Note that γ yields a conjunction between $allocT$ and the encoding of F for existential quantification over reference types, while for universal quantification it yields an implication. Still, the above proof remains the same for existential quantification as well, because the $\mathcal{L}$ operator is defined the same way for conjunction and implication. □

Appendix C

Proof of Theorem 6.2

Theorem 6.2. *If a specification* **Spec** *does not contain recursive preconditions and all of the prescribed proof obligations for* **Spec** *hold, then* **Spec** *is well-formed, that is,* $wf(\mathbf{Spec})$.

We prove a lemma that gives a stronger property than the theorem. From this lemma we trivially get the theorem by Definition 6.2 on page 82.

Lemma. If a specification **Spec** does not contain recursive preconditions and all of the prescribed proof obligations for **Spec** hold, then there is a model $\mathbf{M}$ such that $wf(\mathbf{Spec}, \mathbf{M})$, and such that if $[\mathbf{SysInv}]^3_{\mathbf{M}}\theta$ holds for a variable assignment θ then $(\mathbf{M}, \theta)^{\mathbf{Pre}}$ holds.

Proof. The proof runs by induction on the order in which specification fragments are added. The order is induced by the traversal of the dependency graph. For the proof, we may assume the induction hypothesis, that is, we may assume that the property holds for specification fragments already processed. Formally, we may assume that, there is a model $\mathbf{M}$ such that $wf(\mathbf{Spec}_{j-1}, \mathbf{M})$ and such that if $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}}\theta$ holds for some θ, then $(\mathbf{M}, \theta)^{\mathbf{Pre}_{j-1}}$ holds.

Base case. As base case, we need to prove that there is a structure $\mathbf{M}_0$ such that $wf(\mathbf{Spec}_0, \mathbf{M}_0)$ and such that if $[\mathbf{SysInv}_0]^3_{\mathbf{M}_0}\theta$ holds for some θ, then $(\mathbf{M}_0, \theta)^{\mathbf{Pre}_0}$ holds. Since $\mathbf{Spec}_0 \triangleq \langle \emptyset, \emptyset, \emptyset \rangle$, any structure $\mathbf{M}_0$ with interpretation $I_0 \triangleq \emptyset$ is a model for $\mathbf{Spec}_0$ and $\mathbf{Pre}_0$. Thus, $wf(\mathbf{Spec}_0, \mathbf{M}_0)$ and $(\mathbf{M}_0, \theta)^{\mathbf{Pre}_0}$ trivially holds for every θ.

Induction step. In step j, we pick a set of nodes G_j. We need to prove that there is a structure $\mathbf{M}_j$ such that $wf(\mathbf{Spec}_j, \mathbf{M}_j)$, and such that if $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ holds for some θ, then $(\mathbf{M}_j, \theta)^{\mathbf{Pre}_j}$ holds. To prove the former, we need to show that structure $\mathbf{M}_j$ fulfills the four criteria presented in

Section 6.2. To prove the latter, we need to show that structure $\mathbf{M}_j$ fulfills the two properties presented in Definition 5.1 under the assumption that $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ holds.

To prove these properties, we consider three cases, one for each kind of set of nodes we might process in a step of the model construction. That is, G_j may contain: (1) exactly one invariant, (2) exactly one non-recursively specified function, or (3) a set of recursively specified functions. For the proof, we may assume that the prescribed proof obligations over the specification fragment of G_j hold in all models. Furthermore, we can assume the induction hypothesis: there exists some structure $\mathbf{M}_{j-1}$ such that $wf(\mathbf{Spec}_{j-1}, \mathbf{M}_{j-1})$, and such that if $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$ holds for some θ, then $(\mathbf{M}_{j-1}, \theta)^{\mathbf{Pre}_{j-1}}$ holds.

Case 1. G_j **contains exactly one invariant** $\mathbf{Inv}_l \in \mathbf{INV}$.

In this case we do not process any function symbol, thus in the newly constructed structure $\mathbf{M}_j$, the interpretation of function symbols remain the same as in $\mathbf{M}_{j-1}$. This means that the model of the previous step is left unchanged: $\mathbf{M}_j \triangleq \mathbf{M}_{j-1}$.

First we prove $wf(\mathbf{Spec}_j, \mathbf{M}_j)$, by showing that $\mathbf{M}_j$ has the desired criteria.

Criterion 1. We need to show $\mathbf{wd}([\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta)$. From the induction hypothesis (Criterion 1), we know that $\mathbf{wd}([\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta)$. That is, $\mathbf{SysInv}_{j-1}$ is well-defined in model $\mathbf{M}_{j-1}$, and therefore it evaluates either to **true** or to **false**. We proceed by a case distinction.

Case *i*: $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$ evaluates to **false**.
Since $\mathbf{M}_j \triangleq \mathbf{M}_{j-1}$, $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_j}\theta$ evaluates to **false**, too. Furthermore, by definition we know that:

$$\mathbf{SysInv}_j \;\equiv\; \mathbf{SysInv}_{j-1} \;\wedge\; (\forall\, o.\; alloc(o, OS) \;\Rightarrow\; \mathbf{Inv}_l) \tag{C.1}$$

As $\mathbf{SysInv}_{j-1}$ evaluates to **false** in $\mathbf{M}_j$, $\mathbf{SysInv}_j$ also evaluates to **false** in $\mathbf{M}_j$. Thus, it is well-defined in model $\mathbf{M}_j$: $\mathbf{wd}([\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta)$.

Case *ii*: $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$ evaluates to **true**.
From the induction hypothesis, we know that if $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$ holds, then $(\mathbf{M}_{j-1}, \theta)^{\mathbf{Pre}_{j-1}}$ holds. Thus, we can derive $(\mathbf{M}_{j-1}, \theta)^{\mathbf{Pre}_{j-1}}$. Since $\mathbf{M}_j \triangleq \mathbf{M}_{j-1}$ and $\mathbf{Pre}_j \equiv \mathbf{Pre}_{j-1}$, we can also derive that $(\mathbf{M}_j, \theta)^{\mathbf{Pre}_j}$ holds. This allows us to apply Theorem 5.1 for model $\mathbf{M}_j$ to obtain:

$$[\mathcal{L}(\mathbf{SysInv}_j, \mathbf{Pre}_j)]^2_{\widehat{\mathbf{M}_j}}\theta = \mathbf{wd}([\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta)$$

Thus, it suffices to show that $\widehat{\mathbf{M}}_j$ is model for $\mathcal{L}(\mathbf{SysInv}_j, \mathbf{Pre}_j)$. We proceed in three steps:

1. We derive that for any store OS, $\widehat{\mathbf{M}}_j$ is model for $\mathcal{L}(\mathbf{SysInv}_{j-1}, \mathbf{Pre}_j)$. We know that $\mathbf{wd}([\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta)$ from the induction hypothesis (Criterion 1). As argued above, $(\mathbf{M}_{j-1}, \theta)^{\mathbf{Pre}_{j-1}}$ holds and thus we may apply Theorem 5.1, from which we may conclude that $\widehat{\mathbf{M}}_{j-1}$ is model for $\mathcal{L}(\mathbf{SysInv}_{j-1}, \mathbf{Pre}_{j-1})$. Since $\mathbf{M}_j \triangleq \mathbf{M}_{j-1}$, we know that $\widehat{\mathbf{M}}_j \equiv \widehat{\mathbf{M}}_{j-1}$, too. Furthermore, we know that $\mathbf{Pre}_j \equiv \mathbf{Pre}_{j-1}$. Thus, we can deduce that $\widehat{\mathbf{M}}_j$ is a model for $\mathcal{L}(\mathbf{SysInv}_{j-1}, \mathbf{Pre}_j)$.

2. We derive that for any store OS, $\widehat{\mathbf{M}}_j$ is a model for the formula:

   $$\mathbf{SysInv}_{j-1} \Rightarrow \mathcal{L}(\forall\ o.\ \mathit{alloc}(o, OS) \Rightarrow \mathbf{Inv}_l, \mathbf{Pre}_j)$$

   We can assume that proof obligation (6.3) has been proven. Therefore,

   $$(\forall\ OS.\ \mathbf{SysInv}_{j-1} \Rightarrow \mathcal{L}(\forall\ o.\ \mathit{alloc}(o, OS) \Rightarrow \mathbf{Inv}_l, \mathbf{Pre}_{j-1}))$$

   is known to hold for any total structure. Thus, $\widehat{\mathbf{M}}_j$ is a model for it.

   Since $\mathbf{Pre}_j \equiv \mathbf{Pre}_{j-1}$, we get the property we wanted to derive.

3. We put together the results of the first two steps to derive that $\widehat{\mathbf{M}}_j$ is a model for $\mathcal{L}(\mathbf{SysInv}_j, \mathbf{Pre}_j)$:

   $$\begin{array}{l}\mathcal{L}(\mathbf{SysInv}_{j-1}, \mathbf{Pre}_j) \wedge \\ \qquad (\mathbf{SysInv}_{j-1} \Rightarrow \mathcal{L}(\forall\ o.\ \mathit{alloc}(o, OS) \Rightarrow \mathbf{Inv}_l, \mathbf{Pre}_j)) \ \equiv \\ \mathcal{L}(\mathbf{SysInv}_{j-1} \wedge (\forall\ o.\ \mathit{alloc}(o, OS) \Rightarrow \mathbf{Inv}_l), \mathbf{Pre}_j) \ \equiv \\ \mathcal{L}(\mathbf{SysInv}_j, \mathbf{Pre}_j)\end{array}$$

   In the first two steps we have shown that $\widehat{\mathbf{M}}_j$ is a model for the two conjuncts of the first line.
   The second line is derived by the application of $\mathcal{L}$'s definition on conjunction: $\mathcal{L}(\phi_1 \wedge \phi_2) \triangleq \mathcal{L}(\phi_1) \wedge (\phi_1 \Rightarrow \mathcal{L}(\phi_2))$, with $\phi_1 = \mathbf{SysInv}_{j-1}$, $\phi_2 = \forall\ o.\ \mathit{alloc}(o, OS) \Rightarrow \mathbf{Inv}_l$, and domain restriction $\mathbf{Pre}_j$.
   The third line is derived by equation (C.1), and concludes that $\widehat{\mathbf{M}}_j$ is a model for $\mathcal{L}(\mathbf{SysInv}_j, \mathbf{Pre}_j)$.

This completes Case *ii*, and the proof of Criterion 1.

Criteria 2, 3, and 4. The three criteria contain $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ as guarding condition. Therefore, if we can show that $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ is stronger than $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$, then we trivially get the three criteria by $\mathbf{M}_j \triangleq \mathbf{M}_{j-1}$, and by the induction hypothesis (Criteria 2, 3, and 4).

From Criterion 1 we know that $\mathbf{wd}([\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta)$, and from the induction hypothesis (Criterion 1) that $\mathbf{wd}([\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta)$. That is, both

formulas evaluate to either **true** or **false**. From this fact, and equality (C.1) we can deduce that $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ is stronger than $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$.

This completes the proof of $wf(\mathbf{Spec}_j, \mathbf{M}_j)$ for Case 1.

It remains to prove that if $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ holds for some θ, then $(\mathbf{M}_j, \theta)^{\mathbf{Pre}_j}$ holds. We derive this property as follows:

In Case *ii* for Criterion 1 above, we have already proved that
if $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$ holds, then $(\mathbf{M}_j, \theta)^{\mathbf{Pre}_j}$ holds
As shown above, $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ is stronger than $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$. Thus, we can derive the property we wanted:
if $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ holds for some θ, then $(\mathbf{M}_j, \theta)^{\mathbf{Pre}_j}$ holds.

Case 2. G_j contains exactly one non-recursively specified function $f_l \in \mathbf{F}$.

We construct structure $\mathbf{M}_j$ from $\mathbf{M}_{j-1}$ by adding an interpretation for function f_l. We define the domain of the interpretation of f_l as the set for which the store satisfies the invariants, and the interpretation of $\mathbf{Pre}(f_l)$ in $\mathbf{M}_{j-1}$ is **true**. We define the value of the interpretation of function f_l to be the value of witness $resV$ which satisfies the second conjunct of proof obligation (6.5).

First, we prove $wf(\mathbf{Spec}_j, \mathbf{M}_j)$ by showing that structure $\mathbf{M}_j$ fulfills the four desired criteria.

Criterion 1. We need to show $\mathbf{wd}([\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta)$. From the induction hypothesis (Criterion 1), we know that $\mathbf{wd}([\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta)$ holds. Since $\mathbf{INV}_j \equiv \mathbf{INV}_{j-1}$, we know that $\mathbf{wd}([\mathbf{SysInv}_j]^3_{\mathbf{M}_{j-1}}\theta)$ holds as well. Furthermore, structure $\mathbf{M}_j$ is constructed such that the interpretation of functions on which invariants of $\mathbf{INV}_j$ depend is left unchanged. Thus, we can derive that $\mathbf{wd}([\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta)$ holds.

Criterion 2. We need to show that for each $f \in F_j$, if $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ holds, and **this** is allocated and **par** is alive in **store**, then $\mathbf{wd}([\mathbf{Pre}(f)]^3_{\mathbf{M}_j}\theta)$ holds.

First, we deduce that the criterion holds for all functions in $F_{j-1} = F_j \setminus \{f_l\}$. From the induction hypothesis (Criterion 2), we know that for each $f \in F_{j-1}$, if $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$ holds, then $\mathbf{wd}([\mathbf{Pre}(f)]^3_{\mathbf{M}_{j-1}}\theta)$ holds. Since $\mathbf{INV}_j \equiv \mathbf{INV}_{j-1}$, and structure $\mathbf{M}_j$ is built from $\mathbf{M}_{j-1}$ so that the interpretation of all functions in F_{j-1} is left unchanged, we can derive that $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta = [\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$. Furthermore, we know that $\mathbf{Pre}_j \equiv \mathbf{Pre}_{j-1} \cup \{\langle f_l, \mathbf{Pre}(f_l)\rangle\}$, that is, the domain restrictions of functions in

F_{j-1} did not change. Thus, $\mathbf{wd}([\mathbf{Pre}(f)]^3_{\mathbf{M}_j}\theta) = \mathbf{wd}([\mathbf{Pre}(f)]^3_{\mathbf{M}_{j-1}}\theta)$, for each $f \in F_{j-1}$. Therefore, the criterion holds in $\mathbf{M}_j$ for all functions in F_{j-1}.

It remains to show that the criterion holds for function f_l. We proceed by case distinction.

Case *i*: $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ does not hold. The property trivially holds.

Case *ii*: $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ holds. We need to prove that $\mathbf{wd}([\mathbf{Pre}(f)]^3_{\mathbf{M}_j}\theta)$ holds, which we do in two steps.

1. We derive $\mathbf{wd}([\mathbf{Pre}(f_l)]^3_{\mathbf{M}_j}\theta) = [\mathcal{L}(\mathbf{Pre}(f_l), \mathbf{Pre}_{j-1})]^2_{\widehat{\mathbf{M}}_{j-1}}\theta$.

   Above we have seen that $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta = [\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$, thus we know that $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$ holds. By the induction hypothesis, we know that if $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$ holds then $(\mathbf{M}_{j-1}, \theta)^{\mathbf{Pre}_{j-1}}$ holds. Thus, we can deduce that $(\mathbf{M}_{j-1}, \theta)^{\mathbf{Pre}_{j-1}}$ holds. This allows us to apply Theorem 5.1 and get

   $$[\mathcal{L}(\mathbf{Pre}(f_l), \mathbf{Pre}_{j-1})]^2_{\widehat{\mathbf{M}}_{j-1}}\theta \;=\; \mathbf{wd}([\mathbf{Pre}(f_l)]^3_{\mathbf{M}_{j-1}}\theta)$$

   Since the precondition of f_l does not contain recursive occurrences, $\mathbf{Pre}(f_l)$ does not depend on f_l. Consequently:

   $$\mathbf{wd}([\mathbf{Pre}(f_l)]^3_{\mathbf{M}_{j-1}}\theta) \;=\; \mathbf{wd}([\mathbf{Pre}(f_l)]^3_{\mathbf{M}_j}\theta)$$

   Putting together the two equalities above, we get the desired property.

2. We show that $[\mathcal{L}(\mathbf{Pre}(f_l), \mathbf{Pre}_{j-1})]^2_{\widehat{\mathbf{M}}_{j-1}}\theta$ holds.

   We can assume that proof obligation (6.4) is proven, and thereby that the formula holds for any total structure. Thus, $\widehat{\mathbf{M}}_{j-1}$ is a model for it. By the induction hypothesis, we know that $wf(\mathbf{Spec}_{j-1}, \mathbf{M}_{j-1})$, from which we can deduce that $\mathbf{M}_{j-1} \models \mathbf{Ax}_{\mathbf{Spec}_{j-1}}$. If partial structure $\mathbf{M}_{j-1}$ is a model for $\mathbf{Ax}_{\mathbf{Spec}_{j-1}}$, then the same holds for its extension: $\widehat{\mathbf{M}}_{j-1} \models \mathbf{Ax}_{\mathbf{Spec}_{j-1}}$. Thus, by modus ponens with $\mathbf{Ax}_{\mathbf{Spec}_{j-1}}$ and formula (6.4), we get that $\widehat{\mathbf{M}}_{j-1}$ is a model for:

   $$\begin{aligned}\forall\, OS.\ (&\mathbf{SysInv}_{j-1} \Rightarrow \\ &\forall\, o, p.\ allocT(o, OS, T) \wedge alive(p, OS) \Rightarrow \mathcal{L}(\mathbf{Pre}(f_l), \mathbf{Pre}_{j-1}))\end{aligned} \tag{C.2}$$

   As shown above, $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$ holds. As $\mathbf{SysInv}_{j-1}$ holds for partial structure $\mathbf{M}_{j-1}$, it also holds for $\widehat{\mathbf{M}}_{j-1}$, the total counterpart of $\mathbf{M}_{j-1}$. Therefore, $[\mathbf{SysInv}_{j-1}]^2_{\widehat{\mathbf{M}}_{j-1}}\theta$ holds. From this and the assumptions on the receiver object and the parameter, we can deduce from (C.2) that $\mathcal{L}(\mathbf{Pre}(f_l), \mathbf{Pre}_{j-1})$ holds in $\widehat{\mathbf{M}}_{j-1}$. That is, $[\mathcal{L}(\mathbf{Pre}(f_l), \mathbf{Pre}_{j-1})]^2_{\widehat{\mathbf{M}}_{j-1}}\theta$ holds, and that is what we wanted to prove.

Putting together the results of these two steps, we can deduce that predicate $\mathbf{wd}([\mathbf{Pre}(f_l)]^3_{\mathbf{M}_j}\theta)$ holds, which completes Case *ii*, and the proof of Criterion 2.

Criterion 3. We need to prove that for each $f \in F_j$, if $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ holds, and **this** is allocated and **par** is alive in **store**, and $[\mathbf{Pre}(f)]^3_{\mathbf{M}_j}\theta$ holds, then $\langle \mathbf{store}, \mathbf{this}, \mathbf{par}\rangle \in \mathbf{dom}(\mathbf{I}(f))$ holds.

Since for all $f \in F_{j-1}$ formula $\mathbf{Pre}(f)$ is unchanged, the proof for functions in F_{j-1} is analogous to the previous case. For function f_l, we get the criterion from the way $\mathbf{M}_j$ is constructed.

Criterion 4. We need to prove that for each $f \in F_j$, if $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ holds, and **this** is allocated and **par** is alive in **store**, and $[\mathbf{Pre}(f)]^3_{\mathbf{M}_j}\theta$ holds, then (A) for each **result** that is alive in **store**: $\mathbf{wd}([\mathbf{Post}(f)]^3_{\mathbf{M}_j}\theta')$ holds, and (B) $[\mathbf{Post}(f)]^3_{\mathbf{M}_j}\theta$ holds.

Since for all $f \in F_{j-1}$ formula $\mathbf{Post}(f)$ is unchanged, the proof for functions in F_{j-1} is analogous to that of Criterion 2.

It remains to prove that the criterion holds for f_l. We begin by showing that $\widehat{\mathbf{M}}_{j-1}$ and $\widehat{\mathbf{M}}_j$ are models for formula (C.3) below.

We can assume that proof obligation (6.5) has been proven. Thus, we know that it holds in all total structures, in particular, in $\widehat{\mathbf{M}}_{j-1}$. By the induction hypothesis, we know that $wf(\mathbf{Spec}_{j-1}, \mathbf{M}_{j-1})$ holds, from which we can deduce $\mathbf{M}_{j-1} \models \mathbf{Ax}_{\mathbf{Spec}_{j-1}}$. If partial structure $\mathbf{M}_{j-1}$ is a model for $\mathbf{Ax}_{\mathbf{Spec}_{j-1}}$, then the same holds for its extension too: $\widehat{\mathbf{M}}_{j-1} \models \mathbf{Ax}_{\mathbf{Spec}_{j-1}}$. Thus, by modus ponens on $\mathbf{Ax}_{\mathbf{Spec}_{j-1}}$ and proof obligation (6.5), we can derive that $\widehat{\mathbf{M}}_{j-1}$ is a model for the formula:

$$
\begin{array}{l}
\forall\ OS, o, p. \\
\quad (\mathbf{SysInv}_{j-1} \wedge allocT(o, OS, T)\ \wedge\ alive(p, OS) \wedge\ \mathbf{Pre}(f_l)) \\
\qquad \Rightarrow \\
\quad (\ (\forall\ resV.\ alive(resV, OS) \Rightarrow \mathcal{L}(\mathbf{Post}(f_l), \mathbf{Pre}_{j-1})) \wedge \\
\qquad (\exists\ resV.\ alive(resV, OS) \wedge \mathbf{Post}(f_l))\ )
\end{array}
\tag{C.3}
$$

Next, we show that $\widehat{\mathbf{M}}_j$ is a model for (C.3). As proof obligation (6.5) is assumed to be proven, it holds for any total model, in particular, for $\widehat{\mathbf{M}}_j$. As seen above, $\mathbf{M}_{j-1}$ is a model for $\mathbf{Ax}_{\mathbf{Spec}_{j-1}}$. Since $\mathbf{Ax}_{\mathbf{Spec}_{j-1}}$ does not depend on f_l, $\mathbf{M}_j$ is also a model for it. Thus, as before, by modus ponens, we can deduce that (C.3) holds in model $\widehat{\mathbf{M}}_j$.

We prove (A) and (B) separately, and begin with the proof of (A).

We make a case split on the value of $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$:

Case *i*: $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$ is **false**. In this case, $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ is also **false**, as shown above in the proof of Criterion 2. Thus, the criterion trivially holds.

Case *ii*: $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$ is **true**. We proceed in three steps:

1. We derive that $[\mathbf{SysInv}_{j-1}]^2_{\widehat{\mathbf{M}}_{j-1}}\theta$ and $[\mathbf{Pre}(f_l)]^2_{\widehat{\mathbf{M}}_{j-1}}\theta$ holds.

   Since $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$ is **true**, formula $\mathbf{SysInv}_{j-1}$ is well-defined in model $\mathbf{M}_{j-1}$, and thus we have

   $$[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta \;=\; [\mathbf{SysInv}_{j-1}]^2_{\widehat{\mathbf{M}}_{j-1}}\theta \;=\; \mathbf{true}$$

   The criterion only has to hold under the assumption that $[\mathbf{Pre}(f_l)]^3_{\mathbf{M}_j}\theta$ holds, thus we can assume that formula $\mathbf{Pre}(f_l)$ is well-defined in model $\mathbf{M}_j$. Thus, we have

   $$[\mathbf{Pre}(f_l)]^3_{\mathbf{M}_j}\theta \;=\; [\mathbf{Pre}(f_l)]^2_{\widehat{\mathbf{M}}_j}\theta \;=\; \mathbf{true}$$

   Since the precondition of f_l does not contain recursive occurrences, the evaluation of the precondition does not depend on f_l. Therefore:

   $$[\mathbf{Pre}(f_l)]^3_{\mathbf{M}_j}\theta \;=\; [\mathbf{Pre}(f_l)]^3_{\mathbf{M}_{j-1}}\theta \;=\; \mathbf{true}$$

   This means that $\mathbf{Pre}(f_l)$ is well-defined in model $\mathbf{M}_{j-1}$, too. Consequently, $[\mathbf{Pre}(f_l)]^2_{\widehat{\mathbf{M}}_{j-1}}\theta$ holds.

2. We derive that $[\mathcal{L}(\mathbf{Post}(f_l), \mathbf{Pre}_{j-1})]^2_{\widehat{\mathbf{M}}_{j-1}}\theta' = \mathbf{wd}([\mathbf{Post}(f_l)]^3_{\mathbf{M}_j}\theta')$

   By assumption, $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$ is **true**. Since $\mathbf{SysInv}_{j-1}$ must not mention $resV$, $[\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta'$ is **true**, too. Therefore, we may apply the induction hypothesis with variables assignment θ' to derive that $(\mathbf{M}_{j-1}, \theta')^{\mathbf{Pre}_{j-1}}$ holds. This allows us to apply Theorem 5.1 for model $\mathbf{M}_{j-1}$, predicate $\mathcal{L}(\mathbf{Post}(f_l), \mathbf{Pre}_{j-1})$ from (C.3), and θ' to get

   $$[\mathcal{L}(\mathbf{Post}(f_l), \mathbf{Pre}_{j-1})]^2_{\widehat{\mathbf{M}}_{j-1}}\theta' \;=\; \mathbf{wd}([\mathbf{Post}(f_l)]^3_{\mathbf{M}_{j-1}}\theta').$$

   Since the first conjunct in the consequence of (C.3) holds for all $resV$, in the instantiation of Theorem 5.1 we can use θ' with any **result** that is alive in OS.

   Since the postcondition of f_l does not contain recursive occurrences, the well-definedness of the postcondition does not depend on f_l. Thus, we can deduce

   $$\mathbf{wd}([\mathbf{Post}(f_l)]^3_{\mathbf{M}_j}\theta') \;=\; \mathbf{wd}([\mathbf{Post}(f_l)]^3_{\mathbf{M}_{j-1}}\theta')$$

   From the two equalities above, we get the desired property.

3. We derive (A) using the results of the previous steps and (C.3).

We have already shown that $\widehat{\mathbf{M}}_{j-1}$ is a model for formula (C.3). By the results of the first step and the assumptions on **this** and **par**, we can deduce that $\widehat{\mathbf{M}}_{j-1}$ is also a model for the formula:

$$\forall\ OS, o, p.\ (\forall\ resV.\ alive(resV, OS) \Rightarrow \mathcal{L}(\mathbf{Post}(f_l), \mathbf{Pre}_{j-1}))$$

From this and the equality derived in the second step, we can deduce that $\mathbf{wd}([\mathbf{Post}(f_l)]^3_{\mathbf{M}_j}\theta')$ holds for every **result** that is alive in OS. This is property (A) we wanted to prove.

Next, we prove property (B).

We have seen above that $\widehat{\mathbf{M}}_j$ is a model for formula (C.3). By the assumptions of the criterion on $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$, **this**, **par**, and $[\mathbf{Pre}(f)]^3_{\mathbf{M}_j}\theta$, we deduce from (C.3) that for any OS, o, p, structure $\widehat{\mathbf{M}}_j$ is a model for:

$\exists\ resV.\ alive(resV, OS) \wedge \mathbf{Post}(f_l)$

For Property (A), we have shown that $\mathbf{Post}(f_l)$ is well-defined for any result values in model $\mathbf{M}_j$. Therefore, $\mathbf{M}_j$ is also a model for the above formula. From this and the way we construct the interpretation for f_l (*i.e.*, picking $resV$ as the value that satisfies the existential quantifier), we get property (B).

This completes the proof of $wf(\mathbf{Spec}_j, \mathbf{M}_j)$ for Case 2.

It remains to prove that if $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ holds for some θ, then $(\mathbf{M}_j, \theta)^{\mathbf{Pre}_j}$ holds. Based on Definition 5.1, we need to prove that the following two properties hold:

1. if $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ holds, then for each $f \in F_j$, $\mathbf{wd}([\mathbf{Pre}_j(f)]^3_{\mathbf{M}_j}\theta)$ holds.

   First, we show that the property holds for all symbols $F_{j-1} = F_j \setminus \{f_l\}$. We have shown above that $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta = [\mathbf{SysInv}_{j-1}]^3_{\mathbf{M}_{j-1}}\theta$ and that $\mathbf{wd}([\mathbf{Pre}(f)]^3_{\mathbf{M}_j}\theta) = \mathbf{wd}([\mathbf{Pre}(f)]^3_{\mathbf{M}_{j-1}}\theta)$ for each $f \in F_{j-1}$. Thus, from the induction hypothesis we can derive that the property holds for all functions in F_{j-1}.

   It remains to prove the property for function f_l: if $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ holds then $\mathbf{wd}([\mathbf{Pre}(f_l)]^3_{\mathbf{M}_j}\theta)$. This follows from Criterion 2 we have proven above.

2. if $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ holds, then for each $f \in F_j$ and $\mathbf{val}_1, \ldots, \mathbf{val}_k \in \mathbf{Value}$, $[\mathbf{Pre}_j(f)]^3_{\mathbf{M}}\theta$ holds if and only if $\langle \mathbf{val}_1, \ldots, \mathbf{val}_k \rangle \in \mathbf{dom}(\mathbf{I}(f))$, where $\theta \triangleq [v_1 \rightarrow \mathbf{val}_1, \ldots, v_k \rightarrow \mathbf{val}_k]$ and $\{v_1, \ldots, v_k\}$ are the parameter names of f.

   Analogously to the previous case, it suffices to show that the property

holds for f_l. For function f_l, the property directly follows from the way the interpretation of f_l is defined.

Case 3. G_j **is a set of recursively specified functions** $\{g_1, g_2, \ldots, g_k\}$.

We construct $\mathbf{M}_j$ from $\mathbf{M}_{j-1}$ by adding interpretations for the functions in G_j. We build the desired model recursively. We define depth s of the recursion as the set of input vectors ($\langle \mathbf{store}, \mathbf{this}, \mathbf{par} \rangle$) for which the measure function is equal to s. We denote the model for which the recursion depth is smaller than s for functions in $F_{j-1} \cup G_j$ as $\mathbf{M}^s_{j-1}$. $\mathbf{M}^s_{j-1}$ is a model for the following specification:

$$\mathbf{Spec}^s_{j-1} \triangleq \langle \mathbf{Pre}_{j-1} \cup \{\langle g_l, \mathbf{Pre}(g_l) \wedge \|\langle h, o, p\rangle\|_{g_l} < s\rangle \mid l \in 1..k\}, \mathbf{Post}_j, \mathbf{INV}_j \rangle$$

Note that the preconditions of the functions in G_j, which act as domain restrictions, prevent the postconditions of functions in $\mathbf{M}^s_{j-1}$ from containing function applications with measures greater or equal to s.

Note also that the sets of processed postconditions and invariants do not change along the depth of the recursion. Thus, as in the definition of $\mathbf{Spec}^s_{j-1}$ above, we use $\mathbf{Post}_j$ and $\mathbf{INV}_j$ to refer to these sets along all recursion depths. We apply the same convention for $\mathbf{SysInv}_j$ and F_j. The first component of $\mathbf{Spec}^s_{j-1}$ will be denote by $\mathbf{Pre}^s_{j-1}$.

We define model $\mathbf{M}_j$ as follows: $\mathbf{M}_j \triangleq \mathbf{M}^\infty_{j-1}$. We prove the existence of a model $\mathbf{M}_j$ such that $wf(\mathbf{Spec}_j, \mathbf{M}_j)$ by showing that *for any* s there is a model $\mathbf{M}^{s+1}_{j-1}$ such that $wf(\mathbf{Spec}^{s+1}_{j-1}, \mathbf{M}^{s+1}_{j-1})$. Analogously, we prove that if $[\mathbf{SysInv}_j]^3_{\mathbf{M}_j}\theta$ holds for some θ, then $(\mathbf{M}_j, \theta)^{\mathbf{Pre}_j}$ holds by showing that *for any* s if $[\mathbf{SysInv}_j]^3_{\mathbf{M}^{s+1}_{j-1}}\theta$ holds, then $(\mathbf{M}^{s+1}_{j-1}, \theta)^{\mathbf{Pre}^{s+1}_{j-1}}$ holds. The proof runs by (nested) induction on s.

Base case. In the base case $s = 0$. Since measures are non-negative, $\|h, o, p\|_{g_i} < 0$ is **false** for any $g_i \in G_j$. Thus, the precondition of g_i at $\mathbf{Spec}^0_{j-1}$, which we use as domain restriction, is always false. Therefore, we select the following interpretation to extend $\mathbf{M}_{j-1}$:

$$I^0_{j-1} \triangleq I_{j-1} \cup \{\langle g_l, \emptyset\rangle \mid l \in 1..k\}$$

This ensures that $wf(\mathbf{Spec}^0_{j-1}, \mathbf{M}^0_{j-1})$ and $(\mathbf{M}^0_{j-1}, \theta)^{\mathbf{Pre}^0_{j-1}}$ holds for every θ.

Induction step. For the induction step, we may assume the induction hypothesis: there is a model $\mathbf{M}^s_{j-1}$ such that $wf(\mathbf{Spec}^s_{j-1}, \mathbf{M}^s_{j-1})$ and such that if $[\mathbf{SysInv}_j]^3_{\mathbf{M}^s_{j-1}}\theta$ holds for some θ, then $(\mathbf{M}^s_{j-1}, \theta)^{\mathbf{Pre}^s_{j-1}}$ holds.

We construct $\mathbf{M}_{j-1}^{s+1}$ from $\mathbf{M}_{j-1}^{s}$ by adding interpretations for the functions $g_i \in G_j$ for depth s. To do so, we first construct a new function g_i' that is only defined for s. Then we merge the interpretation of g_i' with the interpretation of function g_i defined in model $\mathbf{M}_{j-1}^{s}$ (that is, defined up to depth $s-1$). The merged function yields function g_i defined in model $\mathbf{M}_{j-1}^{s+1}$ (that is, defined up to depth s). The merge can be done since the merged functions are defined on disjoint domains.

The interpretation of function g_i in model $\mathbf{M}_{j-1}^{s+1}$ is defined as follows. The domain of the interpretation is defined to be the domain of the interpretation of g_i in model $\mathbf{M}_{j-1}^{s}$, extended with the domain of g_i' (*i.e.*, the union of the two domains). The domain of g_i' is defined to be the set of input parameters ($\langle \mathbf{store}, \mathbf{this}, \mathbf{par} \rangle$) for which (*i*) the heap satisfies invariants $\mathbf{INV}_j$, (*ii*) the interpretation of $\mathbf{Pre}(g_i)$ in $\mathbf{M}_{j-1}^{s}$ is **true**, and (*iii*) the measure function is equal to s.

We define the value of the interpretation for function g_i in $\mathbf{M}_{j-1}^{s+1}$ to be witness $resV$ that satisfies the existential quantifier in proof obligation (6.6).

First we prove $wf(\mathbf{Spec}_{j-1}^{s+1}, \mathbf{M}_{j-1}^{s+1})$, by showing that structure $\mathbf{M}_{j-1}^{s+1}$ fulfills the four desired criteria.

Criterion 1. We need to show $\mathbf{wd}([\mathbf{SysInv}_j]^3_{\mathbf{M}_{j-1}^{s+1}}\theta)$. As the set of invariants remains the same, the proof is analogous to the proof of Criterion 1 for Case 2.

Criterion 2. We need to show that for each $f \in F_j$, if $[\mathbf{SysInv}_j]^3_{\mathbf{M}_{j-1}^{s+1}}\theta$ holds, then $\mathbf{wd}([\mathbf{Pre}_{j-1}^{s+1}(f)]^3_{\mathbf{M}_{j-1}^{s+1}}\theta)$ holds.

First, we deduce that the criterion holds for all functions in $F_{j-1} = F_j \setminus G_j$. The proof is analogous to the corresponding proof of Criterion 2 for Case 2.

It remains to prove that the criterion holds for all functions $g_i \in G_j$. From the way $\mathbf{Spec}_{j-1}^{s+1}$ is constructed, we know for all functions $g_i \in G_j$:

$\mathbf{Pre}_{j-1}^{s+1}(g_i) \equiv \mathbf{Pre}(g_i) \wedge \|\langle h, o, p \rangle\|_{g_i} < s+1$

Thus, we have to prove that the conjunct is well-defined in structure $\mathbf{M}_{j-1}^{s+1}$. $\mathbf{Pre}(g_i)$ is required to be non-recursive, thus the proof of $\mathbf{wd}([\mathbf{Pre}(g_i)]^3_{\mathbf{M}_{j-1}^{s+1}}\theta)$ is analogous to the corresponding proof of Criterion 2 for Case 2.

From $\mathbf{wd}([\mathbf{Pre}(g_i)]^3_{\mathbf{M}_{j-1}^{s+1}}\theta)$ we know that $\mathbf{Pre}(g_i)$ is well-defined in structure $\mathbf{M}_{j-1}^{s+1}$. Therefore, we can deduce that it evaluates either to **true** or to **false**. We make a case distinction on the evaluation.

Case *i*: $\mathbf{Pre}(g_i)$ evaluates to **false**. In this case, $\mathbf{Pre}_{j-1}^{s+1}(g_i)$ also evaluates to **false**, therefore it is well-defined.

Case *ii*: $\mathbf{Pre}(g_i)$ evaluates to **true**. In this case, we can apply the induction hypothesis (Criterion 4) and deduce that the measure function of g_i is well-defined. This follows from the way the dependency graph is built in the presence of measure functions (see Page 94). Thus, we can deduce that $\|\langle h, o, p\rangle\|_{g_i}$ is well-defined in structure $\mathbf{M}_{j-1}^{s+1}$. Since $<$ and $+$ are total operators and s is a natural number, we can deduce that $\|\langle h, o, p\rangle\|_{g_i} < s+1$ is also well-defined. Since both operands of the conjunct are well-defined in structure $\mathbf{M}_{j-1}^{s+1}$, we can conclude that their conjunction is also well-defined in the structure.

Criterion 3. We need to prove that for each $f \in F_j$, if $[\mathbf{SysInv}_j]^3_{\mathbf{M}_{j-1}^{s+1}}\theta$ holds, and **this** is allocated and **par** is alive in **store**, and $[\mathbf{Pre}_{j-1}^{s+1}(f)]^3_{\mathbf{M}_{j-1}^{s+1}}\theta$ holds, then $\langle \mathbf{store}, \mathbf{this}, \mathbf{par}\rangle \in \mathbf{dom}(\mathbf{I}(f))$ holds.

For all functions in $F_{j-1} = F_j \setminus G_j$, the proof is analogous to the corresponding proof of Criterion 3 for Case 2. For functions in G_j, we trivially get the criterion from the way structure $\mathbf{M}_{j-1}^{s+1}$ is constructed.

Criterion 4. We need to prove that for each $f \in F_j$, if $[\mathbf{SysInv}_j]^3_{\mathbf{M}_{j-1}^{s+1}}\theta$ and $[\mathbf{Pre}_{j-1}^{s+1}(f)]^3_{\mathbf{M}_{j-1}^{s+1}}\theta$ hold, then (A) for each **result** that is alive in **store**: $\mathbf{wd}([\mathbf{Post}(f)]^3_{\mathbf{M}_{j-1}^{s+1}}\theta')$ holds, and (B) $[\mathbf{Post}(f)]^3_{\mathbf{M}_{j-1}^{s+1}}\theta$ holds.

Since for all $f \in F_{j-1}$ formula $\mathbf{Post}(f)$ is unchanged, the proof is analogous to the corresponding proof of Criterion 4 for Case 2.

It remains to prove that the criterion holds for functions in G_j. In proof obligation (6.6), the depth of the recursion is determined by the value of variable *ind*. We can assume that the proof obligation holds for all *ind*.

Let us consider the instantiation for $ind = s$:

$$
\begin{array}{l}
\mathbf{Ax}_{\mathbf{Spec}_{j-1}} \Rightarrow \\
\quad \forall\ OS, o, p. \\
\quad\ (\ \mathbf{SysInv}_{j-1} \wedge allocT(o, OS, T) \wedge alive(p, OS) \wedge \mathbf{Pre}(g_i) \wedge \|\langle o, p, OS\rangle\|_{g_i} = s \wedge \\
\qquad (\ \bigwedge_{l=1}^{k} \forall\ o', p'.\ allocT(o', OS, T_{g_l}) \wedge alive(p', OS) \wedge \mathbf{Pre}(g_l)[o'/o, p'/p] \wedge \\
\qquad\qquad \|\langle o', p', OS\rangle\|_{g_l} < s \Rightarrow \mathbf{Post}(g_l)[o'/o, p'/p, g_l(o', p', OS)/resV]\ )\ ) \\
\qquad \Rightarrow \\
\quad (\ (\forall\ resV.\ alive(resV, OS) \Rightarrow \\
\qquad \mathcal{L}(\mathbf{Post}(g_i), \mathbf{Pre}_{j-1} \cup \{\langle g_l, \mathbf{Pre}(g_l) \wedge \|\langle o, p, OS\rangle\|_{g_l} < s\rangle \mid l \in 1..k\})) \wedge \\
\quad\ (\exists\ resV.\ alive(resV, OS) \wedge \mathbf{Post}(g_i))\ )
\end{array}
\tag{C.4}
$$

The formula holds in all models, in particular, in $\widehat{\mathbf{M}}_{j-1}^{s}$. From the induction hypothesis running over j, we know that $\mathbf{Ax}_{\mathbf{Spec}_{j-1}}$ holds for the model. From the induction hypothesis running over s, we know that all functions $g_i \in G_j$ are well-formed for recursion depth smaller than s. This means that Criterion 4 holds for $\mathbf{M}_{j-1}^{s}$ for functions g_i defined up to depth

smaller than s. From this we can conclude[1] that lines 4 and 5 of the above formula hold for $\widehat{\mathbf{M}}^{s}_{j-1}$. Furthermore, from the definition of the recursion depth, we know that conjunct $\|\langle h, o, p\rangle\|_{g_i} = s$ holds in induction step s. These facts allow us to reduce the above formula to a formula similar in form to that of (C.3) in the proof of Criterion 4 for Case 2.

We also know that the precondition does not contain recursive occurrences. Furthermore, we know that the postcondition does not contain recursive occurrences, since we are constructing a new function g'_i. Thus, we can apply analogous reasoning to what we applied to show Criterion 4 in Case 2. The only difference in the reasoning is that in this case it is applied over s and not j. That is, it is applied over the nested induction hypothesis.

This allows us to deduce that Criterion 4 carries over from model $\mathbf{M}^{s}_{j-1}$ to model $\mathbf{M}^{s+1}_{j-1}$.

This completes the proof of $wf(\mathbf{Spec}^{s+1}_{j-1}, \mathbf{M}^{s+1}_{j-1})$ as well as the proof of $wf(\mathbf{Spec}_j, \mathbf{M}_j)$ for the non-nested induction step.

It remains to prove that if $[\mathbf{SysInv}_j]^3_{\mathbf{M}^{s+1}_{j-1}}\theta$ holds, then predicate $(\mathbf{M}^{s+1}_{j-1}, \theta)^{\mathbf{Pre}^{s+1}_{j-1}}$ holds. We need to prove that the following two properties hold:

1. if $[\mathbf{SysInv}_j]^3_{\mathbf{M}^{s+1}_{j-1}}\theta$ holds then for each $f \in F_j$, $\mathbf{wd}([\mathbf{Pre}^{s+1}_{j-1}(f)]^3_{\mathbf{M}^{s+1}_{j-1}}\theta)$ holds.

   Analogously to Case 2, we can derive that the criterion holds for all functions $F_{j-1} = F_j \setminus G_j$.

   It remains to prove that if $[\mathbf{SysInv}_j]^3_{\mathbf{M}^{s+1}_{j-1}}\theta$ holds, then for each $g_i \in G_j$, $\mathbf{wd}([\mathbf{Pre}^{s+1}_{j-1}(g_i)]^3_{\mathbf{M}^{s+1}_{j-1}}\theta)$ holds. This follows from Criterion 2 we have proven above.

2. if $[\mathbf{SysInv}_j]^3_{\mathbf{M}^{s+1}_{j-1}}\theta$ holds then for each function symbol $f \in F_j$ and $\mathbf{val}_1, \ldots, \mathbf{val}_k \in \mathbf{Value}$, $[\mathbf{Pre}^{s+1}_{j-1}(f)]^3_{\mathbf{M}^{s+1}_{j-1}}\theta$ holds if and only if $\langle \mathbf{val}_1, \ldots, \mathbf{val}_k\rangle \in \mathbf{dom}(\mathbf{I}(f))$, where $e \triangleq [v_1 \rightarrow \mathbf{val}_1, \ldots, v_k \rightarrow \mathbf{val}_k]$ and $\{v_1, \ldots, v_k\}$ are the parameter names of f.

   Analogously to the previous case, it suffices to show that the criterion holds for all functions $g_i \in G_j$. For functions in G_j, the criterion directly follows from the way the interpretations of the functions are defined.

□

[1] Note that the form of lines 4 and 5 is slightly different than the form of Criterion 4. However, the two forms are equivalent since $\mathbf{SysInv}_{j-1}$ is present in the antecedent (on line 3) of proof obligation (C.4).

Bibliography

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2009. To be published.

[3] J.-R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, 77(1–2):1–28, 2007.

[4] J.-R. Abrial and L. Mussat. On using conditional definitions in formal theories. In *Formal Specification and Development in Z and B (ZB'02)*, pages 242–269. Springer-Verlag, 2002.

[5] S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, 2000.

[6] R. D. Arthan. Undefinedness in Z: Issues for specification and proof. In M. Kerber, editor, *CADE-13 Workshop on Mechanization of Partial Functions*, pages 3–12, Rutgers University, New Brunswick, NJ, 1996.

[7] J. Barnes. *High Integrity Software: The* SPARK *Approach to Safety and Security*. Addison-Wesley, 2003.

[8] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO)*, volume 4111 of *LNCS*, pages 364–387. Springer-Verlag, 2005.

[9] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology (JOT)*, 3(6):27–56, 2004.

[10] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of*

Safe, Secure and Interoperable Smart devices (CASSIS), volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.

[11] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. Allowing state changes in specifications. In G. Müller, editor, *Emerging Trends in Information and Communication Security (ETRICS)*, volume 3995 of *LNCS*, pages 321–336. Springer-Verlag, 2006.

[12] C. W. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker category B. In R. Alur and D. Peled, editors, *CAV*, volume 3114 of *LNCS*, pages 515–518. Springer-Verlag, 2004.

[13] H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[14] B. Beckert, M. Giese, E. Habermalz, R. Hähnle, A. Roth, P. Rümmer, and S. Schlager. Taclets: A new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas (RACSAM)*, 98(1), 2004.

[15] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.

[16] P. Behm, L. Burdy, and J.-M. Meynadier. Well defined B. In *International B Conference*, volume 1393 of *LNCS*, pages 29–45. Springer-Verlag, 1998.

[17] S. Berezin, C. Barrett, I. Shikanian, M. Chechik, A. Gurfinkel, and D. L. Dill. A practical approach to partial functions in CVC Lite. *Electronic Notes in Theoretical Computer Science*, 125(3):13–23, 2005.

[18] S. Berghofer. Definitorische Konstruktion induktiver Datentypen in Isabelle/HOL. Master's thesis, Institut für Informatik, Technische Universität München, 1998. (In German).

[19] S. Berghofer and M. Wenzel. Inductive datatypes in HOL – Lessons learned in formal-logic engineering. In *Theorem Proving in Higher Order Logics (TPHOLs)*, pages 19–36. Springer-Verlag, 1999.

[20] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[21] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, 1995.

[22] C.-B. Breunesse and E. Poll. Verifying JML specifications with model fields. In *Formal Techniques for Java-like Programs (FTfJP)*, pages 51–60, 2003. Technical Report 408, ETH Zurich.

[23] A. D. Brucker and B. Wolff. The HOL-OCL Book. Technical Report 525, ETH Zürich, 2006.

[24] A. D. Brucker and B. Wolff. Semantics, calculi, and analysis for object-oriented specifications. *Acta Informatica*, 2009.

[25] R. Bubel, R. Hähnle, and P. H. Schmitt. Specification predicates with explicit dependency information. In B. Beckert, editor, *Verification Workshop (VERIFY'08)*, volume 372 of *CEUR Workshop Proceedings*, pages 28–43. CEUR-WS.org, 2008.

[26] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Formal Methods Europe (FME)*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, 2003.

[27] P. Chalin. Are the logical foundations of verifying compiler prototypes matching user expectations? *Formal Aspects of Computing*, 19(2):139–158, 2007.

[28] P. Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *International Conference on Software Engineering (ICSE)*, pages 23–33. IEEE Computer Society, 2007.

[29] J. Charles. Adding native specifications to JML. In E. Zucca and D. Ancona, editors, *Formal Techniques for Java-like Programs (FTfJP)*, 2006.

[30] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *Refinement Workshop*, pages 51–69. Springer-Verlag, 1991.

[31] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software: Practice and Experience*, 35(6):583–599, 2005.

[32] D. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology (JOT)*, 4(8):77–103, 2005.

[33] Á. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In M. B. Dwyer and A. Lopes, editors, *Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *LNCS*, pages 336–351. Springer-Verlag, 2007.

[34] Á. Darvas and P. Müller. Formal encoding of JML Level 0 specifications in JIVE. In B. Meyer, P. Müller, and M. Oriol, editors, *Software Engineering 2007*. Technical Report 559, ETH Zurich, 2007.

[35] Á. Darvas and P. Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology (JOT)*, 5(5):59–85, 2006.

[36] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *LNCS*, pages 337–340. Springer-Verlag, 2008.

[37] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[38] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, SRC HP Labs, 2003.

[39] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *International Conference on Software Engineering (ICSE)*, pages 258–267. IEEE Computer Society Press, 1996.

[40] E. W. Dijkstra. Structured programming. In *Software Engineering Techniques*. NATO Science Committee, 1970.

[41] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[42] W. M. Farmer. Theory interpretation in simple type theory. In *Higher-Order Algebra, Logic, and Term Rewriting*, volume 816 of *LNCS*, pages 96–123. Springer-Verlag, 1994.

[43] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Technical Report 1366, LRI, Université Paris Sud, 2003.

[44] J.-C. Filliâtre, T. Hubert, and C. Marché. The Caduceus verification tool for C programs. Tutorial and Reference Manual, 2007.

[45] J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In E. Clarke, G. Agha, and D. Kroening, editors, *International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *LNCS*, pages 15–29. Springer-Verlag, 2004.

[46] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Computer Aided Verification (CAV)*, LNCS, pages 173–177. Springer-Verlag, 2007.

[47] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI)*, volume 37, pages 234–245. ACM Press, 2002.

[48] M. Gordon, A. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *LNCS*. Springer-Verlag, 1979.

[49] D. Gries. *The Science of Programming*. Monographs in Computer Science. Springer-Verlag, 1987.

[50] D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*, pages 366–373. Springer-Verlag, 1995.

[51] J. V. Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, 1977.

[52] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[53] R. Hähnle. *Automated Deduction in Multiple-Valued Logics*, volume 10 of *International Series of Monographs on Computer Science*. Oxford University Press, 1994.

[54] R. Hähnle. Advanced many-valued logics. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 2, pages 297–395. Kluwer, Dordrecht, 2nd edition, 2001.

[55] R. Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IGPL*, 13(4):415–433, 2005.

[56] J. G. Hall, J. A. McDermid, and I. Toyn. Model conjectures for Z specifications. In *Putting into Practice Methods and Tools for Information System Design*, pages 41–51, 1995.

[57] N. Hamilton, R. Nickson, O. Traynor, and M. Utting. Interpretation and instantiation of theories for reasoning about formal specifications. In *Australasian Computer Science Conference (ACSC), Australian Computer Science Communications 19*, pages 37–45, 1997.

[58] D. Harel. Dynamic logic. In *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, chapter 10, pages 497–604. Reidel, Dordrecht, 1984.

[59] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, 2000.

[60] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In L. Cardelli, editor, *European Conference on Object-Oriented Programming (ECOOP)*, LNCS, pages 431–456. Springer-Verlag, 2003.

[61] J. Henkel and A. Diwan. A tool for writing and debugging algebraic specifications. In *International Conference on Software Engineering (ICSE)*, pages 449–458. IEEE Computer Society, 2004.

[62] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[63] A. Hoogewijs. On a formalization of the non-definedness notion. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 25:213–217, 1979.

[64] T. Hubert and C. Marché. A case study of C source code verification: the Schorr-Waite algorithm. In B. K. Aichernig and B. Beckert, editors, *Software Engineering and Formal Methods (SEFM)*, pages 190–199. IEEE Computer Society Press, 2005.

[65] B. Jacobs and F. Piessens. Inspector methods for state abstraction. *Journal of Object Technology (JOT)*, 6(5):5–28, 2007.

[66] Java™ Platform, Standard Edition 6, API Specification, 2008.

[67] JML Distribution. `http://sourceforge.net/projects/jmlspecs`. The model library resides in package `org.jmlspecs.models`.

[68] C. B. Jones. *Systematic software development using VDM*. Prentice Hall, 1986.

[69] I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Formal Methods (FM)*, volume 4085 of *LNCS*, pages 268–283. Springer-Verlag, 2006.

[70] J. R. Kiniry and D. R. Cok. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, volume 3362 of *LNCS*, pages 108–128. Springer-Verlag, 2005.

[71] S. C. Kleene. On a notation for ordinal numbers. *Journal of Symbolic Logic*, 3:150–155, 1938.

[72] G. T. Leavens. Larch/C++ an interface specification language for C++. Department of Computer Science, Iowa State University, 2007. ftp://ftp.cs.iastate.edu/pub/larchc++/poster.ps.gz.

[73] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[74] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[75] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1–3):185–205, 2005.

[76] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[77] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. JML Reference Manual, Revision 1.235, 2008/07/17. Department of Computer Science, Iowa State University.

[78] G. T. Leavens and J. M. Wing. Protective interface specifications. *Formal Aspects of Computing*, 10(1):59–75, 1998.

[79] K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995.

[80] K. R. M. Leino. Data groups: specifying the modification of extended state. *SIGPLAN Notices*, 33(10):144–153, 1998.

[81] K. R. M. Leino. Specification and verification of object-oriented software. In M. Broy, W. Sitou, and T. Hoare, editors, *Engineering Methods and Tools for Software Safety and Security*, volume 22 of *NATO Science for Peace and Security Series*, pages 231–266. IOS Press, 2009.

[82] K. R. M. Leino and R. Middelkoop. Proving consistency of pure methods and model fields. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE)*, volume 5503 of *LNCS*, pages 231–245. Springer-Verlag, 2009.

[83] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.

[84] K. R. M. Leino and P. Müller. A verification methodology for model fields. In P. Sestoft, editor, *European Symposium on Programming (ESOP)*, volume 3924 of *LNCS*, pages 115–130. Springer-Verlag, 2006.

[85] K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In S. Drossopoulou, editor, *European Symposium on Programming (ESOP)*, volume 4960 of *LNCS*, pages 307–321. Springer-Verlag, 2008.

[86] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, 2002.

[87] K. R. M. Leino and W. Schulte. A verifying compiler for a multi-threaded object-oriented language. In *Software System Reliability and Security*, pages 351–416. IOS Press, 2007.

[88] B. Levy. *An Approach to Compiler Correctness Using Interpretation Between Theories*. PhD thesis, University of California, Los Angeles, 1986.

[89] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[90] C. Marché. Towards modular algebraic specifications for pointer programs: A case study. In *Rewriting, Computation and Proof*, volume 4600 of *LNCS*, pages 235–258. Springer-Verlag, 2007.

[91] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58:89–106, 2004.

[92] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[93] T. F. Melham. Automating recursive type definitions in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

[94] B. Meyer. Design by Contract. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice Hall, 1991.

[95] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[96] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[98] J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description. Unpublished, 2000. http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=MeyerMuellerPoetzsch-Heffter00.pdf.

[99] M. Miragliotta. Specification model library for the interactive program prover JIVE. ETH Zurich, Semester Thesis, 2004. http://www.pm.inf.ethz.ch/education/theses/student_docs/Marcello_Miragliotta/Marcello_Miragliotta_paper.pdf.

[100] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[101] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[102] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[103] D. A. Naumann. Observational purity and encapsulation. *Theoretical Computer Science*, 376(3):205–224, 2007.

[104] R. Nickson, O. Traynor, and M. Utting. Cogito ergo sum: Providing structured theorem prover support for specification formalisms. In *Australasian Computer Science Conference (ACSC)*, pages 149–158, 1996.

[105] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing, 1992.

[106] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[107] T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle's logics: HOL, 2009. http://isabelle.in.tum.de/doc/logics-HOL.pdf.

[108] I. Nunes, A. Lopes, V. T. Vasconcelos, J. Abreu, and L. S. Reis. Checking the conformance of Java classes against algebraic specifications. In Z. Liu and J. He, editors, *International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *LNCS*, pages 494–513. Springer-Verlag, 2006.

[109] Object Constraint Language, OMG Available Specification, Version 2.0, 2006. http://www.omg.org/docs/formal/06-05-01.pdf.

[110] S. Owre and N. Shankar. A brief overview of PVS. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *LNCS*, pages 22–27. Springer-Verlag, 2008.

[111] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods (PROCOMET)*, pages 404–423, 1998.

[112] A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report 00-03a, Department of Computer Science, Iowa State University, 2000.

[113] T. RAISE Method Group. *The RAISE Development Method*. The BCS Practitioners Series. Prentice-Hall, 1995.

[114] A. Roth. Specification and verification of encapsulation in Java programs. In M. Steffen and G. Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 3535 of *LNCS*, pages 195–210. Springer-Verlag, 2005.

[115] A. Roth and P. H. Schmitt. Ensuring invariant contracts for modules in Java. In *Formal Techniques for Java-like Programs (FTfJP)*, pages 93–102, 2004.

[116] A. Rudich, Á. Darvas, and P. Müller. Checking well-formedness of pure-method specifications. In *Formal Methods (FM)*, volume 5014 of *LNCS*, pages 68–83. Springer-Verlag, 2008.

[117] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[118] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *LNCS*, pages 199–215. Springer-Verlag, 2005.

[119] S. Sankar. Run-time consistency checking of algebraic specifications. In *Symposium on Testing, Analysis, and Verification*, pages 123–129. ACM Press, 1991.

[120] B. Schieder and M. Broy. Adapting calculational logic to the undefined. *The Computer Journal*, 42(2):73–81, 1999.

[121] B. Schoeller. Strengthening Eiffel contracts using models. In H. D. Van and Z. Liu, editors, *Proceedings of the Workshop on Formal Aspects of Component Software (FACS)*, pages 143–158, 2003. Number 284 in UNU/IIST Reports.

[122] B. Schoeller. *Making Classes Provable Through Contracts, Models and Frames*. PhD thesis, ETH Zurich, 2007.

[123] B. Schoeller, T. Widmer, and B. Meyer. Making specifications complete through models. In R. H. Reussner, J. A. Stafford, and C. A. Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, pages 48–70. Springer-Verlag, 2006.

[124] J. Shield, I. Hayes, and D. Carrington. Using theory interpretation to mechanise the reals in a theorem prover. *Electronic Notes in Theoretical Computer Science*, 42:266–281, 2001.

[125] J. R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

[126] J. Smans, B. Jacobs, and F. Piessens. VeriCool: An automatic verifier for a concurrent object-oriented language. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 5051 of *LNCS*, pages 220–239. Springer-Verlag, 2008.

[127] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In S. Drossopoulou, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 5653 of *LNCS*. Springer-Verlag, 2009.

[128] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *Fundamental Approaches to Software Engineering (FASE)*, volume 4961 of *LNCS*, pages 261–275. Springer-Verlag, 2008.

[129] J. M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1988.

[130] G. Sutcliffe, C. B. Suttner, and T. Yemenis. The TPTP problem library. In *Conference on Automated Deduction (CADE)*, volume 814 of *LNCS*, pages 252–266. Springer-Verlag, 1994.

[131] OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.2, 2009. http://www.omg.org/docs/formal/09-02-04.pdf.

[132] S. H. Valentine. Inconsistency and undefinedness in Z - A practical guide. In *International Conference of Z Users*, volume 1493 of *LNCS*, pages 233–249. Springer-Verlag, 1998.

[133] G. von Roten. Proving well-formedness of interface specifications. Master's thesis, ETH Zurich, 2007.

[134] J. M. Wing. *A Two-Tiered Approach to Specifying Programs*. PhD thesis, Massachusetts Institute of Technology, 1983.

[135] J. M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, 1987.

[136] Sources and Isabelle proofs of the case studies presented in the thesis. www.pm.inf.ethz.ch/people/darvasa/thesis-casestudies.tar.

Curriculum Vitae

Personal Data

Name:	Ádám Péter Darvas
Date of Birth:	February 18, 1979
Nationality:	Hungarian
E-mail:	adam.darvas@gmail.com

Education and Employment History

10/2003 – 9/2009: *ETH Zürich, Switzerland*
Department of Computer Science, Chair of Programming Methodology
Doctoral Student, Research and Teaching Assistant

6/2006 – 9/2006: *Microsoft Research, Redmond, WA, USA*
Summer Intern at the Group of Programming Languages and Methods

11/2002 – 6/2003: *Chalmers University of Technology, Gothenburg, Sweden*
Department of Computer Science and Engineering
Research and Teaching Assistant

9/1997 – 7/2002: *Budapest University of Technology and Economics, Hungary*
Department of Electrical Engineering and Informatics
Diploma in Technical Informatics

Selected Publications

Á. Darvas and P. Müller: Faithful Mapping of Model Classes to Mathematical Structures. *IET Software*, 2(6):477–499, 2008.

Á. Darvas, F. Mehta, and A. Rudich: Efficient Well-Definedness Checking. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 5195 of *LNCS*, pages 100–115. Springer-Verlag, 2008.

A. Rudich, Á. Darvas, and P. Müller: Checking Well-Formedness of Pure-Method Specifications. In *Formal Methods (FM)*, volume 5014 of *LNCS*, pages 68–83. Springer-Verlag, 2008.

Á. Darvas and K. R. M. Leino: Practical Reasoning About Invocations and Implementations of Pure Methods. In *Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *LNCS*, pages 336–351. Springer-Verlag, 2007.

Á. Darvas and P. Müller: Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology (JOT)*, 5(5):59–85, 2006.

Á. Darvas, R. Hähnle, and D. Sands: A Theorem Proving Approach to Analysis of Secure Information Flow. In *International Conference on Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193–209. Springer-Verlag, 2005.

I. Majzik, Á. Darvas, and B. Benyó: Verification of UML Statechart Models of Embedded Systems. In *IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS)*, 2002.